

QNX® Neutrino® RTOS

C Library Reference



©1996–2014, QNX Software Systems Limited, a subsidiary of BlackBerry. All rights reserved.

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: <http://www.qnx.com/>

QNX, QNX CAR, Neutrino, Momentics, Aviage, and Foundry27 are trademarks of BlackBerry Limited that are registered and/or used in certain jurisdictions, and used under license by QNX Software Systems Limited. All other trademarks belong to their respective owners.

Electronic edition published: Monday, March 17, 2014

Table of Contents

About This Reference	41
Typographical conventions	42
Technical support	44
 Chapter 1: What's in a Function Description?	 45
 Chapter 2: Manifests	 51
 Chapter 3: A	 53
abort()	54
abs()	56
accept()	58
access()	61
acl_add_perm()	64
acl_calc_mask()	66
acl_clear_perms()	68
acl_copy_entry()	70
acl_copy_ext()	72
acl_copy_int()	74
acl_create_entry()	76
acl_delete_entry()	78
acl_delete_perm()	80
acl_dup()	82
acl_free()	84
acl_from_text()	86
acl_get_entry()	88
acl_get_fd()	90
acl_get_file()	92
acl_get_permset()	94
acl_get_qualifier()	96
acl_get_tag_type()	98
acl_init()	100
acl_set_fd()	102
acl_set_file()	104
acl_set_permset()	106
acl_set_qualifier()	108
acl_set_tag_type()	110
acl_size()	112
acl_to_text()	114
acl_valid()	116

acos(), acosf(), acosl()	118
acosh(), acoshf(), acoshl()	120
addrinfo	122
aio_cancel()	124
aio_error()	127
aio_fsync()	129
aio_read(), aio_read64()	131
aio_return()	134
aio_suspend()	136
aio_write(), aio_write64()	138
aiocb, aiocb64	141
alarm()	144
alloca()	147
alphasort()	149
_amblksiz	151
_argc	153
_argv	154
asctime(), asctime_r()	155
asin(), asinf(), asinl()	157
asinh(), asinhf(), asinhl()	159
assert()	161
atan(), atanf(), atanl()	163
atan2(), atan2f()	165
atanh(), atanhf(), atanhhl()	167
atexit()	169
atof()	171
atoh()	173
atoi()	175
atol(), atoll()	177
atomic_add()	179
atomic_add_value()	181
atomic_clr()	183
atomic_clr_value()	185
atomic_set()	187
atomic_set_value()	189
atomic_sub()	191
atomic_sub_value()	193
atomic_toggle()	195
atomic_toggle_value()	197
_auxv	199
Chapter 4: B	201
basename()	202
bcmp()	204

bcopy()	206
bind()	208
bindresvport()	211
brk()	213
bsearch()	216
bt_get_backtrace()	219
bt_init_accessor()	221
bt_load_memmap()	223
bt_release_accessor()	225
bt_set_flags()	227
bt_sprn_memmap()	229
bt_sprnf_addrs()	231
bt_translate_addrs()	234
bt_unload_memmap()	236
_btext	237
btowc()	238
bzero()	239
 Chapter 5: C	 241
cabs(), cabsf()	242
cache_fini()	244
CACHE_FLUSH()	246
cache_init()	248
CACHE_INVALID()	253
calloc()	255
cbrt(), cbrtf()	258
ceil(), ceilf()	260
cfgetispeed()	262
cfgetospeed()	264
cfgopen()	266
cfmakeraw()	270
cfree()	272
cfsetispeed()	274
cfsetospeed()	276
ChannelCreate(), ChannelCreate_r()	278
ChannelDestroy(), ChannelDestroy_r()	284
chdir()	286
chmod()	289
chown()	292
chroot()	295
chsize()	298
clearenv()	300
clearerr()	302
ClientInfoExtFree()	304

clock()	305
clock_getcpuclockid()	307
clock_getres()	309
clock_gettime()	312
clock_nanosleep()	315
clock_settime()	319
ClockAdjust(), ClockAdjust_r()	322
ClockCycles()	325
ClockId(), ClockId_r()	327
ClockPeriod(), ClockPeriod_r()	329
ClockTime(), ClockTime_r()	333
close()	336
closedir()	338
closelog()	340
_cmdfd()	341
_cmdname()	342
confstr()	344
connect()	348
ConnectAttach(), ConnectAttach_r()	351
ConnectClientInfo(), ConnectClientInfo_r()	356
ConnectClientInfoAble()	360
ConnectClientInfoExt()	363
ConnectDetach(), ConnectDetach_r()	367
ConnectFlags(), ConnectFlags_r()	369
ConnectServerInfo(), ConnectServerInfo_r()	372
copysign(), copysignf()	375
cos(), cosf(), cosl()	377
cosh(), coshf(), coshl()	379
creat(), creat64()	381
crypt()	385
ctermid()	387
ctime(), ctime_r()	389
Chapter 6: D	393
daemon()	394
daylight	396
DebugBreak()	397
DebugKDBreak()	398
DebugKDOutput()	399
delay()	401
devctl()	403
dhcpcctl*()	412
difftime()	417
dirctl()	419

dirent, dirent64	422
dirname()	424
dispatch_block()	426
dispatch_context_alloc()	429
dispatch_context_free()	431
dispatch_create()	433
dispatch_create_channel()	436
dispatch_destroy()	438
dispatch_handler()	440
dispatch_timeout()	443
dispatch_unblock()	445
div()	447
dladdr()	449
dlclose()	451
dlderror()	453
dlopen()	454
dlsym()	461
dn_comp()	464
dn_expand()	466
drand48()	468
ds_clear()	469
ds_create()	471
ds_deregister()	474
ds_flags()	476
ds_get()	478
ds_register()	480
ds_set()	481
dup()	483
dup2()	485
 Chapter 7: E	487
eaccess()	488
_edata	491
encrypt()	492
_end	494
endfsent()	495
endgrent()	496
endhostent()	497
ENDIAN_BE16()	498
ENDIAN_BE32()	500
ENDIAN_BE64()	502
ENDIAN_LE16()	504
ENDIAN_LE32()	506
ENDIAN_LE64()	508

ENDIAN_RET16()	510
ENDIAN_RET32()	512
ENDIAN_RET64()	514
ENDIAN_SWAP16()	516
ENDIAN_SWAP32()	518
ENDIAN_SWAP64()	520
endnetent()	522
endprotoent()	523
endpwent()	524
endservent()	525
endspent()	526
endutent()	527
environ	528
eof()	529
erand48()	531
erf(), erff()	532
erfc(), erfcf(), erfcl()	534
err(), errx()	536
errno	539
_etext	549
execl()	550
execle()	556
execlp()	562
execlepe()	568
execv()	573
execve()	579
execvp()	585
execvpe()	591
_exit()	596
exit()	599
exp(), expf(), expl()	601
expm1(), expm1f()	603
Chapter 8: F	605
fabs(), fabsf()	606
fcfgopen()	608
fchdir()	610
fchmod()	612
fchown()	615
fclose()	618
fcloseall()	621
fcntl()	623
fdatasync()	631
fdistrusted()	633

fdopen()	635
feof()	637
ferror()	639
fflush()	641
ffs()	644
fgetc()	645
fgetchar()	648
fgetpos()	650
fgets()	652
fgetspent()	654
fgetwc()	656
fgetws()	658
fileno()	661
flink()	663
flock()	666
flockfile()	669
floor(), floorf()	671
flushall()	673
fmod(), fmodf(), fmodl()	675
fnmatch()	677
fopen(), fopen64()	681
fork()	685
forkpty()	688
fp_exception_mask()	690
fp_exception_value()	693
fp_precision()	695
fp_rounding()	697
fpathconf()	699
fprintf()	702
fputc()	704
fputchar()	707
fputs()	709
fputwc()	711
fputws()	713
fread()	715
free()	717
freeaddrinfo()	719
freeifaddrs()	720
freopen(), freopen64()	721
frexp(), frexpf()	725
fs_crypto_check()	727
fs_crypto_domain_add()	729
fs_crypto_domain_key_change()	732
fs_crypto_domain_key_check()	735
fs_crypto_domain_key_size()	738

fs_crypto_domain_lock()	739
fs_crypto_domain_query()	741
fs_crypto_domain_remove()	743
fs_crypto_domain_unlock()	745
fs_crypto_enable()	748
fs_crypto_enable_option()	750
fs_crypto_file_get_domain()	753
fs_crypto_key_gen()	755
fs_crypto_migrate_control()	757
fs_crypto_migrate_status()	760
fs_crypto_migrate_path()	762
fs_crypto_migrate_tag()	764
fs_crypto_file_set_domain()	766
fs_crypto_set_logging()	769
fscanf()	770
fseek(), fseeko(), fseeko64()	772
fsetpos()	776
fstat(), fstat64()	779
fstatvfs(), fstatvfs64()	783
fsync()	787
ftell(), ftello(), ftello64()	789
ftime()	791
ftruncate(), ftruncate64()	793
ftrylockfile()	796
ftw(), ftw64()	798
funlockfile()	801
futime()	802
fwide()	804
fwprintf()	806
fwrite()	808
fwscanf()	810
Chapter 9: G	813
gai_strerror()	814
gamma(), gamma_r(), gammaf(), gammaf_r()	816
getaddrinfo()	818
getc()	823
getc_unlocked()	825
getchar()	827
getchar_unlocked()	829
getcwd()	830
getdomainname()	833
getdtablesize()	835
getegid()	836

getenv()	838
geteuid()	840
getfsfile()	842
getfsent()	844
getfsspec()	847
getgid()	849
getgrent()	851
getgrgid()	853
getgrgid_r()	855
getgrnam()	857
getgrnam_r()	859
getgrouplist()	861
getgroups()	864
gethostbyaddr()	866
gethostbyaddr_r()	869
gethostbyname(), gethostbyname2()	872
gethostbyname_r()	875
gethostent()	878
gethostent_r()	880
gethostname()	883
getifaddrs()	885
GETIOVBASE()	887
GETIOVLEN()	888
getitimer()	889
getlogin()	891
getlogin_r()	892
getnameinfo()	894
getnetbyaddr()	898
getnetbyname()	900
getnetent()	902
getopt()	903
getpagesize()	908
getpagesizes(), getpagesizes64()	909
getpass()	911
getpeereid()	913
getpeername()	915
getpgid()	917
getpgrp()	919
getpid()	920
getppid()	921
getprio()	922
getprotobyname()	924
getprotobynumber()	926
getprotoent()	928
getpwent()	930

getpwent_r()	932
getpwnam()	934
getpwnam_r()	936
getpwuid()	938
getpwuid_r()	940
getrlimit(), getrlimit64()	942
getrusage()	945
gets()	950
getservbyname()	952
getservbyport()	954
getservent()	956
getsid()	958
getsockname()	960
getsockopt()	962
getspent(), getspent_r()	971
getspnam(), getspnam_r()	974
getsubopt()	977
gettid()	981
gettimeofday()	982
getuid()	984
getutent()	985
getutid()	987
getutline()	989
getw()	991
getwc()	993
getwchar()	995
getwd()	997
glob()	999
globfree()	1004
gmtime()	1005
gmtime_r()	1007
grantpt()	1009
Chapter 10: H	1011
h_errno	1012
hcreate()	1014
hdestroy()	1016
herror()	1017
hostent	1019
hsearch()	1021
hstrerror()	1024
htonl()	1026
htons()	1027
hwi_find_item()	1028

hwi_find_tag()	1030
hwi_next_item()	1032
hwi_next_tag()	1035
hwi_off2tag()	1037
hwi_tag2off()	1039
hypot(), hypotf()	1040
Chapter 11: I	1043
ICMP	1045
ICMP6	1047
if_freenameindex()	1049
if_indextoname()	1050
if_nameindex()	1052
if_nametoindex()	1054
ifaddrs	1055
ilogb(), ilogbf()	1057
in8()	1059
in8s()	1061
in16(), inbe16(), inle16()	1063
in16s()	1065
in32(), inbe32(), inle32()	1067
in32s()	1069
index()	1071
inet_addr()	1073
inet_aton()	1075
inet_lnaof()	1077
inet_makeaddr()	1078
inet_net_ntop()	1080
inet_netof()	1083
inet_net_pton()	1085
inet_network()	1087
inet_ntoa()	1089
inet_ntoa_r()	1091
inet_ntop()	1093
inet_pton()	1096
INET6	1100
inet6_option_alloc()	1103
inet6_option_append()	1105
inet6_option_find()	1107
inet6_option_init()	1109
inet6_option_next()	1111
inet6_option_space()	1113
inet6_rthdr_add()	1115
inet6_rthdr_getaddr()	1117

inet6_rthdr_getflags()	1119
inet6_rthdr_init()	1121
inet6_rthdr_lasthop()	1123
inet6_rthdr_reverse()	1125
inet6_rthdr_segments()	1127
inet6_rthdr_space()	1129
initgroups()	1131
initstate()	1133
inotify_add_watch()	1135
inotify_event	1137
inotify_init()	1139
inotify_rm_watch()	1141
input_line()	1143
insque()	1145
InterruptAttach(), InterruptAttach_r()	1147
InterruptAttachEvent(), InterruptAttachEvent_r()	1154
InterruptCharacteristic(), InterruptCharacteristic_r()	1159
InterruptDetach(), InterruptDetach_r()	1162
InterruptDisable()	1164
InterruptEnable()	1166
InterruptHookIdle()	1168
InterruptHookTrace()	1171
InterruptLock()	1174
InterruptMask()	1176
InterruptStatus()	1178
InterruptUnlock()	1179
InterruptUnmask()	1181
InterruptWait(), InterruptWait_r()	1183
_intr_v86()	1185
_io_connect	1189
_io_connect_ftype_reply	1195
_io_connect_link_reply	1196
ioctl(), ioctl_socket()	1198
iofdinfo()	1210
iofunc_acl()	1212
iofunc_acl_default()	1215
iofunc_attr_init()	1217
iofunc_attr_lock()	1219
iofunc_attr_t	1221
iofunc_attr_trylock()	1226
iofunc_attr_unlock()	1228
iofunc_check_access()	1230
iofunc_chmod()	1234
iofunc_chmod_default()	1237
iofunc_chown()	1239

iofunc_chown_default()	1242
iofunc_client_info_able()	1244
iofunc_client_info_ext()	1246
iofunc_client_info_ext_free()	1248
iofunc_close_dup()	1249
iofunc_close_dup_default()	1251
iofunc_close_ocb()	1253
iofunc_close_ocb_default()	1255
iofunc_devctl()	1257
iofunc_devctl_default()	1261
iofunc_devctl_verify()	1263
iofunc_fdinfo()	1266
iofunc_fdinfo_default()	1268
iofunc_func_init()	1271
iofunc_link()	1273
iofunc_lock()	1276
iofunc_lock_calloc()	1279
iofunc_lock_default()	1281
iofunc_lock_free()	1283
iofunc_lock_ocb_default()	1284
iofunc_lseek()	1286
iofunc_lseek_default()	1289
iofunc_mknod()	1291
iofunc_mmap()	1294
iofunc_mmap_default()	1298
iofunc_notify()	1300
iofunc_notify_remove()	1306
iofunc_notify_trigger()	1308
iofunc_ocb_attach()	1311
iofunc_ocb_calloc()	1313
iofunc_ocb_detach()	1315
iofunc_ocb_free()	1317
iofunc_ocb_t	1319
iofunc_open()	1321
iofunc_open_default()	1325
iofunc_openfd()	1327
iofunc_openfd_default()	1330
iofunc_pathconf()	1332
iofunc_pathconf_default()	1335
iofunc_read_default()	1337
iofunc_read_verify()	1339
iofunc_readlink()	1343
iofunc_rename()	1346
iofunc_space_verify()	1350
iofunc_stat()	1353

iofunc_stat_default()	1355
iofunc_sync()	1358
iofunc_sync_default()	1360
iofunc_sync_verify()	1362
iofunc_time_update()	1364
iofunc_unblock()	1366
iofunc_unblock_default()	1368
iofunc_unlink()	1370
iofunc_unlock_ocb_default()	1373
iofunc_utime()	1375
iofunc_utime_default()	1378
iofunc_write_default()	1380
iofunc_write_verify()	1382
ionotify()	1386
IP	1391
IPsec	1396
ipsec_dump_policy()	1402
ipsec_get_policylen()	1404
ipsec_set_policy()	1406
ipsec_strerror()	1410
IPv6	1412
iruserok(), iruserok_sa()	1419
isalnum()	1421
isalpha()	1423
isascii()	1425
isatty()	1427
isblank()	1429
iscntrl()	1431
isdigit()	1433
isfdtype()	1435
isfinite()	1437
isgraph()	1439
isinf()	1441
islower()	1443
isnan()	1445
isprint()	1447
ispunct()	1449
isspace()	1451
isupper()	1453
iswalnum()	1455
iswalpha()	1457
iswblank()	1459
iswcntrl()	1461
iswctype()	1463
iswdigit()	1465

iswgraph()	1467
iswlower()	1469
iswprint()	1471
iswpunct()	1473
iswspace()	1475
iswupper()	1477
iswxdigit()	1479
isxdigit()	1481
itoa()	1483
Chapter 12: J	1485
j0(), j0f()	1486
j1(), j1f()	1488
jn(), jnf()	1490
jrand48()	1492
Chapter 13: K	1493
kill()	1494
killpg()	1496
Chapter 14: L	1499
labs()	1500
lchown()	1502
lcong48()	1505
ldexp(), ldexpf(), ldexpl()	1506
ldiv(), lldiv()	1508
lfind()	1510
lgamma(), lgamma_r(), lgammaf(), lgammaf_r()	1513
link()	1516
lio_listio(), lio_listio64()	1519
listen()	1523
llabs()	1525
localeconv()	1526
localtime()	1531
localtime_r()	1533
lockf(), lockf64()	1535
log(), logf(), logl()	1539
log10(), log10f(), log10l()	1541
log1p(), log1pf(), log1pl()	1543
logb(), logbf()	1545
login_tty()	1547
longjmp()	1549
lrand48()	1551
lsearch()	1552

lseek(), lseek64()	1555
lstat(), lstat64()	1558
ltoa(), lltoa()	1560
ltrunc()	1562
Chapter 15: M	1565
main()	1566
mallinfo()	1568
malloc()	1569
mallopt()	1575
max()	1582
mblen()	1583
mbrlen()	1585
mbrtowc()	1587
mbsinit()	1589
mbsrtowcs()	1591
mbstowcs()	1593
mbtowc()	1595
mcheck()	1598
mem_offset(), mem_offset64()	1600
memalign()	1603
memccpy()	1605
memchr()	1607
memcmp()	1609
memcpy()	1611
memcpyv()	1613
memicmp()	1616
memmove()	1618
memset()	1620
memset_s()	1622
message_attach()	1624
message_connect()	1630
message_detach()	1633
min()	1635
mkdir()	1636
mkfifo()	1639
mknod()	1642
mkstemp()	1645
mktemp()	1647
mktime()	1649
mlock()	1651
mlockall()	1653
mmap(), mmap64()	1656
mmap_device_io()	1666

mmap_device_memory()	1668
modem_open()	1672
modem_read()	1676
modem_script()	1679
modem_write()	1686
modf(), modff()	1689
mount()	1691
mount_parse_generic_args()	1694
mprobe()	1697
mprotect()	1699
mq_close()	1702
mq_getattr()	1704
mq_notify()	1707
mq_open()	1709
mq_receive()	1714
mq_send()	1717
mq_setattr()	1720
mq_timedreceive(), mq_timedreceive_monotonic()	1722
mq_timedsend(), mq_timedsend_monotonic()	1725
mq_unlink()	1728
mrnd48()	1730
_msg_info	1731
MsgCurrent(), MsgCurrent_r()	1734
MsgDeliverEvent(), MsgDeliverEvent_r()	1736
MsgError(), MsgError_r()	1743
MsgInfo(), MsgInfo_r()	1745
MsgKeyData(), MsgKeyData_r()	1747
MsgRead(), MsgRead_r()	1754
MsgReadv(), MsgReadv_r()	1757
MsgReceive(), MsgReceive_r()	1760
MsgReceivePulse(), MsgReceivePulse_r()	1765
MsgReceivePulsev(), MsgReceivePulsev_r()	1768
MsgReceivev(), MsgReceivev_r()	1771
MsgReply(), MsgReply_r()	1775
MsgReplyv(), MsgReplyv_r()	1778
MsgSend(), MsgSend_r()	1781
MsgSendnc(), MsgSendnc_r()	1785
MsgSendPulse(), MsgSendPulse_r()	1789
MsgSendsv(), MsgSendsv_r()	1793
MsgSendsvnc(), MsgSendsvnc_r()	1797
MsgSendv(), MsgSendv_r()	1801
MsgSendvnc(), MsgSendvnc_r()	1805
MsgSendvs(), MsgSendvs_r()	1809
MsgSendvsnc(), MsgSendvsnc_r()	1813
MsgVerifyEvent(), MsgVerifyEvent_r()	1817

MsgWrite(), MsgWrite_r()	1819
MsgWritev(), MsgWritev_r()	1822
msync()	1825
munlock()	1828
munlockall()	1830
munmap()	1832
munmap_device_io()	1834
munmap_device_memory()	1836
munmap_flags()	1838
 Chapter 16: N	 1841
name_attach()	1842
name_close()	1848
name_detach()	1850
name_open()	1852
nanosleep()	1855
nanospin()	1857
nanospin_calibrate()	1859
nanospin_count()	1861
nanospin_ns()	1863
nanospin_ns_to_count()	1865
nap()	1867
napms()	1868
nbaconnect()	1869
nbaconnect_result()	1871
ncurses	1873
ND_NODE_CMP()	1874
netent	1876
netmgr_ndtostr()	1877
netmgr_remote_nd()	1881
netmgr_strtond()	1883
nextafter(), nextafterf()	1885
nftw(), nftw64()	1888
nice()	1892
nrnd48()	1894
nsec2timespec()	1895
ntohl()	1896
ntohs()	1897
 Chapter 17: O	 1899
offsetof()	1900
OMAPI	1902
open(), open64()	1905
opendir()	1913

openfd()	1916
openlog()	1918
openpty()	1921
out8()	1923
out8s()	1925
out16(), outbe16(), outle16()	1927
out16s()	1929
out32(), outbe32(), outle32()	1931
out32s()	1933
 Chapter 18: P	 1935
pathconf()	1936
pathfind(), pathfind_r()	1939
pathmgr_symlink()	1943
pathmgr_unlink()	1945
pause()	1947
pccard_arm()	1949
pccard_attach()	1952
pccard_detach()	1954
pccard_info()	1956
pccard_lock()	1958
pccard_raw_read()	1960
pccard_unlock()	1962
pci_attach()	1964
pci_attach_device()	1966
pci_detach()	1975
pci_detach_device()	1977
pci_find_class()	1979
pci_find_device()	1981
pci_irq_routing_options()	1983
pci_map_irq()	1985
pci_present()	1987
pci_read_config()	1989
pci_read_config8()	1991
pci_read_config16()	1993
pci_read_config32()	1995
pci_rescan_bus()	1997
pci_write_config()	1999
pci_write_config8()	2001
pci_write_config16()	2003
pci_write_config32()	2005
pclose()	2007
perror()	2009
pipe()	2011

poll()	2013
popen()	2019
posix_fadvise(), posix_fadvise64()	2022
posix_fallocate(), posix_fallocate64()	2025
posix_madvise()	2028
posix_mem_offset(), posix_mem_offset64()	2030
posix_memalign()	2033
posix_openpt()	2035
posix_spawn(), posix_spawnnp()	2037
posix_spawn_file_actions_addclose()	2048
posix_spawn_file_actions_adddup2()	2050
posix_spawn_file_actions_addopen()	2052
posix_spawn_file_actions_destroy()	2055
posix_spawn_file_actions_init()	2057
posix_spawnattr_addpartid()	2059
posix_spawnattr_addpartition()	2061
posix_spawnattr_destroy()	2064
posix_spawnattr_getcred()	2066
posix_spawnattr_getflags()	2068
posix_spawnattr_getnode()	2070
posix_spawnattr_getpartid()	2072
posix_spawnattr_getpgroup()	2074
posix_spawnattr_getrunmask()	2076
posix_spawnattr_getschedparam()	2078
posix_spawnattr_getschedpolicy()	2080
posix_spawnattr_getsigdefault()	2082
posix_spawnattr_getsigignore()	2084
posix_spawnattr_getsigmask()	2086
posix_spawnattr_getstackmax()	2088
posix_spawnattr_getxflags()	2090
posix_spawnattr_init()	2093
posix_spawnattr_setcred()	2095
posix_spawnattr_setflags()	2098
posix_spawnattr_setnode()	2101
posix_spawnattr_setpgroup()	2103
posix_spawnattr_setrunmask()	2105
posix_spawnattr_setschedparam()	2107
posix_spawnattr_setschedpolicy()	2109
posix_spawnattr_setsigdefault()	2111
posix_spawnattr_setsigignore()	2113
posix_spawnattr_setsigmask()	2115
posix_spawnattr_setstackmax()	2117
posix_spawnattr_setxflags()	2119
posix_typed_mem_get_info()	2122
posix_typed_mem_open()	2124

pow(), powf(), powl()	2129
pread(), pread64()	2131
printf()	2133
procmgr_ability()	2141
procmgr_ability_create()	2152
procmgr_ability_lookup()	2154
procmgr_daemon()	2156
procmgr_event_notify()	2158
procmgr_event_notify_add()	2163
procmgr_event_notify_delete()	2168
procmgr_event_trigger()	2170
procmgr_guardian()	2172
procmgr_session()	2175
procmgr_timer_tolerance()	2178
procmgr_value_current()	2180
procmgr_value_notify_add()	2181
__progname	2183
protoent	2184
pthread_abort()	2185
pthread_atfork()	2187
pthread_attr_destroy()	2189
pthread_attr_getdetachstate()	2191
pthread_attr_getguardsize()	2193
pthread_attr_getinheritsched()	2195
pthread_attr_getschedparam()	2197
pthread_attr_getschedpolicy()	2199
pthread_attr_getscope()	2201
pthread_attr_getstack()	2203
pthread_attr_getstackaddr()	2205
pthread_attr_getstacklazy()	2207
pthread_attr_getstackprealloc()	2209
pthread_attr_getstacksize()	2211
pthread_attr_init()	2213
pthread_attr_setdetachstate()	2215
pthread_attr_setguardsize()	2217
pthread_attr_setinheritsched()	2219
pthread_attr_setschedparam()	2221
pthread_attr_setschedpolicy()	2223
pthread_attr_setscope()	2225
pthread_attr_setstack()	2227
pthread_attr_setstackaddr()	2229
pthread_attr_setstacklazy()	2231
pthread_attr_setstackprealloc()	2233
pthread_attr_setstacksize()	2235
pthread_barrier_destroy()	2237

pthread_barrier_init()	2239
pthread_barrier_wait()	2241
pthread_barrierattr_destroy()	2243
pthread_barrierattr_getpshared()	2245
pthread_barrierattr_init()	2247
pthread_barrierattr_setpshared()	2249
pthread_cancel()	2251
pthread_cleanup_pop()	2253
pthread_cleanup_push()	2255
pthread_cond_broadcast()	2257
pthread_cond_destroy()	2259
pthread_cond_init()	2261
pthread_cond_signal()	2263
pthread_cond_timedwait()	2265
pthread_cond_wait()	2269
pthread_condattr_destroy()	2272
pthread_condattr_getclock()	2274
pthread_condattr_getpshared()	2276
pthread_condattr_init()	2278
pthread_condattr_setclock()	2280
pthread_condattr_setpshared()	2282
pthread_create()	2284
pthread_detach()	2288
pthread_equal()	2290
pthread_exit()	2292
pthread_getconcurrency()	2294
pthread_getcpuclockid()	2295
pthread_getname_np()	2297
pthread_getschedparam()	2299
pthread_getspecific()	2301
pthread_join()	2303
pthread_key_create()	2305
pthread_key_delete()	2308
pthread_kill()	2310
pthread_mutex_destroy()	2312
pthread_mutex_getprioceiling()	2314
pthread_mutex_init()	2316
pthread_mutex_lock()	2318
pthread_mutex_setprioceiling()	2322
pthread_mutex_timedlock(), pthread_mutex_timedlock_monotonic()	2324
pthread_mutex_trylock()	2327
pthread_mutex_unlock()	2329
pthread_mutex_wakeup_np()	2331
pthread_mutexattr_destroy()	2333
pthread_mutexattr_getprioceiling()	2335

pthread_mutexattr_getprotocol()	2337
pthread_mutexattr_getpshared()	2339
pthread_mutexattr_getrecursive()	2341
pthread_mutexattr_gettype()	2343
pthread_mutexattr_getwakeup_np()	2345
pthread_mutexattr_init()	2347
pthread_mutexattr_setprioceiling()	2349
pthread_mutexattr_setprotocol()	2351
pthread_mutexattr_setpshared()	2353
pthread_mutexattr_setrecursive()	2355
pthread_mutexattr_settype()	2357
pthread_mutexattr_setwakeup_np()	2359
pthread_once()	2361
pthread_rwlock_destroy()	2364
pthread_rwlock_init()	2366
pthread_rwlock_rdlock()	2369
pthread_rwlock_timedrdlock()	2371
pthread_rwlock_timedwrlock()	2373
pthread_rwlock_tryrdlock()	2375
pthread_rwlock_trywrlock()	2377
pthread_rwlock_unlock()	2379
pthread_rwlock_wrlock()	2381
pthread_rwlockattr_destroy()	2383
pthread_rwlockattr_getpshared()	2385
pthread_rwlockattr_init()	2387
pthread_rwlockattr_setpshared()	2389
pthread_self()	2391
pthread_setcancelstate()	2392
pthread_setcanceltype()	2394
pthread_setconcurrency()	2396
pthread_setname_np()	2398
pthread_setschedparam()	2400
pthread_setschedprio()	2402
pthread_setspecific()	2405
pthread_sigmask()	2407
pthread_sleepon_broadcast()	2409
pthread_sleepon_lock()	2411
pthread_sleepon_signal()	2413
pthread_sleepon_timedwait()	2415
pthread_sleepon_unlock()	2418
pthread_sleepon_wait()	2419
pthread_spin_destroy()	2422
pthread_spin_init()	2424
pthread_spin_lock()	2426
pthread_spin_trylock()	2428

pthread_spin_unlock()	2430
pthread_testcancel()	2432
pthread_timedjoin(), pthread_timedjoin_monotonic()	2433
ptsname()	2436
ptsname_r()	2438
_pulse	2440
pulse_attach()	2442
pulse_detach()	2445
putc()	2447
putc_unlocked()	2449
putchar()	2451
putchar_unlocked()	2453
putenv()	2454
puts()	2457
putspent()	2459
pututline()	2462
putw()	2464
putwc()	2466
putwchar()	2468
pwrite(), pwrite64()	2470
 Chapter 19: Q	 2473
qnx_crypt()	2474
qsort()	2476
 Chapter 20: R	 2479
Raccept()	2480
raise()	2482
rand()	2484
rand_r()	2486
random()	2487
Rbind()	2489
rcmd(), rcmd_af()	2491
Rconnect()	2494
rdchk()	2496
re_comp()	2498
re_exec()	2500
read()	2502
readblock()	2506
readcond()	2509
readdir(), readdir64()	2514
readdir_r()	2518
readlink()	2520
readv()	2523

realloc()	2527
realpath()	2531
recv()	2533
recvfrom()	2536
recvmsg()	2539
regcomp()	2542
regerror()	2547
regexec()	2549
regfree()	2552
remainder(), remainderf(), remainderl()	2554
remove()	2556
remque()	2559
rename()	2560
res_init()	2563
res_mkquery()	2566
res_query()	2569
res_querydomain()	2572
res_search()	2575
res_send()	2578
resmgr_attach()	2580
resmgr_block()	2589
resmgr_connect_funcs_t	2592
resmgr_context_alloc()	2594
resmgr_context_free()	2596
resmgr_context_t	2598
resmgr_detach()	2600
resmgr_devino()	2603
resmgr_handle_grow()	2605
resmgr_handle_tune()	2606
resmgr_handler()	2608
resmgr_io_funcs_t	2610
resmgr_iofuncs()	2614
resmgr_msg_again()	2616
resmgr_msgread()	2618
resmgr_msgreadv()	2620
resmgr_msgreply()	2622
resmgr_msgreplyv()	2625
resmgr_msgwrite()	2628
resmgr_msgwritev()	2630
_RESMGR_NPARTS()	2632
resmgr_ocb()	2634
resmgr_open_bind()	2636
resmgr_pathname()	2639
_RESMGR_PTR()	2641
_RESMGR_STATUS()	2643

resmgr_unbind()	2644
rewind()	2646
rewinddir()	2648
Rgetsockname()	2650
rindex()	2652
rint(), rintf(), rintl()	2654
Rlisten()	2656
rmdir()	2658
round(), roundf(), roundl()	2661
ROUTE	2663
Rrcmd()	2668
rresvport(), rresvport_af()	2670
Rselect()	2672
rsrdbmgr_attach()	2674
rsrdbmgr_create()	2681
rsrdbmgr_destroy()	2684
rsrdbmgr_detach()	2686
rsrdbmgr_devno_attach()	2688
rsrdbmgr_devno_detach()	2691
rsrdbmgr_query_name()	2693
ruserok()	2696
Chapter 21: S	2699
sbrk()	2702
scalb()	2705
scalbn(), scalbnf()	2707
_scalloc()	2709
scandir()	2711
scanf()	2713
sched_getparam()	2720
sched_get_priority_adjust()	2723
sched_get_priority_max()	2725
sched_get_priority_min()	2727
sched_getscheduler()	2729
sched_param	2731
sched_rr_get_interval()	2736
sched_setparam()	2738
sched_setscheduler()	2740
sched_yield()	2743
SchedCtl(), SchedCtl_r()	2745
SchedGet(), SchedGet_r()	2774
SchedInfo(), SchedInfo_r()	2777
SchedSet(), SchedSet_r()	2780
SchedYield(), SchedYield_r()	2783

searchenv()	2785
seed48()	2787
seekdir()	2789
select()	2791
select_attach()	2796
select_detach()	2800
select_query()	2802
sem_close()	2805
sem_destroy()	2807
sem_getvalue()	2809
sem_init()	2811
sem_open()	2814
sem_post()	2818
sem_timedwait(), sem_timedwait_monotonic()	2820
sem_trywait()	2823
sem_unlink()	2825
sem_wait()	2827
send()	2829
sendmsg()	2833
sendto()	2837
servent	2842
set_lowest_fd()	2843
setbuf()	2845
setbuffer()	2847
setdomainname()	2849
setegid()	2851
setenv()	2853
seteuid()	2856
setgid()	2858
setfsent()	2860
setgrent()	2862
setgroups()	2864
sethostent()	2866
sethostname()	2868
SETIOV()	2870
setitimer()	2872
setjmp()	2875
setkey()	2877
setlinebuf()	2878
setlocale()	2880
setlogmask()	2882
setnetent()	2884
setpgid()	2886
setpgrp()	2889
setprio()	2890

setprotoent()	2892
setpwent()	2894
setregid()	2895
setreuid()	2897
setrlimit(), setrlimit64()	2899
setservent()	2905
setsid()	2907
setsockopt()	2909
setspent()	2912
setstate()	2913
settimeofday()	2915
setuid()	2917
setutent()	2919
setvbuf()	2920
_sfree()	2923
shm_ctl()	2925
shm_ctl_special()	2931
shm_open()	2935
shm_unlink()	2940
shutdown()	2942
shutdown_system()	2944
sigaction()	2948
sigaddset()	2953
sigblock()	2955
sigdelset()	2957
sigemptyset()	2959
sigevent	2961
sigfillset()	2966
sigismember()	2968
siglongjmp()	2970
sigmask()	2972
signal()	2974
SignalAction(), SignalAction_r()	2977
SignalKill(), SignalKill_r()	2986
SignalProcmask(), SignalProcmask_r()	2991
SignalSuspend(), SignalSuspend_r()	2995
SignalWaitinfo(), SignalWaitinfo_r()	2997
sigpause()	3000
sigpending()	3002
sigprocmask()	3004
sigqueue()	3007
sigsetjmp()	3010
sigsetmask()	3012
sigsuspend()	3014
sigtimedwait()	3016

sigunblock()	3018
sigwait()	3020
sigwaitinfo()	3022
sin(), sinf(), sinl()	3024
sinh(), sinh(), sinhl()	3026
sleep()	3028
_sleepon_broadcast()	3030
_sleepon_destroy()	3032
_sleepon_init()	3034
_sleepon_lock()	3036
_sleepon_signal()	3038
_sleepon_unlock()	3040
_sleepon_wait()	3042
slog2c()	3044
slog2f()	3046
slog2fa()	3048
slog2_get_verbosity()	3051
slog2_register()	3052
slog2_reset()	3056
slog2_set_default_buffer()	3057
slog2_set_verbosity()	3059
slogb()	3061
slogf()	3063
slogi()	3066
_smlloc()	3068
snprintf()	3070
socketatmark()	3073
socket()	3075
socketpair()	3078
SOCKSinit()	3081
sopen()	3083
sopenfd()	3087
spawn()	3089
spawnl()	3099
spawnle()	3105
spawnlp()	3111
spawnlpe()	3116
spawnp()	3122
spawnv()	3129
spawnve()	3134
spawnvp()	3140
spawnvpe()	3145
sprintf()	3151
sqrt(), sqrtf(), sqrtl()	3153
srand()	3155

srand48()	3157
srandom()	3158
_srealloc()	3160
sscanf()	3162
stat(), stat64()	3164
statvfs(), statvfs64()	3170
stderr	3175
stdin	3176
stdout	3177
straddstr()	3178
strcasecmp()	3180
strcat()	3182
strchr()	3184
strcmp()	3186
strcmpi()	3188
strcoll()	3190
strcpy()	3192
strcspn()	3194
strdup()	3196
strerror()	3198
strerror_r()	3200
strftime()	3202
stricmp()	3208
strlcat(), strlcpy()	3210
strlen()	3213
strlwr()	3215
strncasecmp()	3217
strncat()	3219
strncmp()	3221
strncpy()	3223
strnicmp()	3225
strnlen()	3227
strnset()	3229
strpbrk()	3231
strptime()	3233
strrchr()	3239
strrev()	3241
strsep()	3243
strset()	3245
strsignal()	3247
strspn()	3249
strstr()	3251
strtod(), strtod(), strtold()	3253
strtoimax(), strtoumax()	3256
strtok()	3258

strtok_r()	3260
strtol(), strtoll()	3262
strtoul(), strtoull()	3264
strupr()	3266
strxfrm()	3268
swab()	3270
swprintf()	3272
swscanf()	3274
symlink()	3276
sync()	3279
SyncCondvarSignal(), SyncCondvarSignal_r()	3280
SyncCondvarWait(), SyncCondvarWait_r()	3282
SyncCtl(), SyncCtl_r()	3286
SyncDestroy(), SyncDestroy_r()	3289
SyncMutexEvent(), SyncMutexEvent_r()	3291
SyncMutexLock(), SyncMutexLock_r()	3294
SyncMutexRevive(), SyncMutexRevive_r()	3297
SyncMutexUnlock(), SyncMutexUnlock_r()	3299
SyncSemPost(), SyncSemPost_r()	3301
SyncSemWait(), SyncSemWait_r()	3303
SyncTypeCreate(), SyncTypeCreate_r()	3305
sysconf()	3308
sysctl()	3311
syslog()	3316
sysmgr_reboot()	3319
sysmgr_runstate(), sysmgr_runstate_dynamic()	3320
sysmgr_runstate_burst()	3322
SYSPAGE_CPU_ENTRY()	3324
SYSPAGE_ENTRY()	3326
_syspage_ptr	3329
system()	3330
Chapter 22: T	3333
tan(), tanf(), tanl()	3334
tanh(), tanhf(), tanhl()	3336
tcdrain()	3338
tcdropline()	3340
tcflow()	3342
tcflush()	3345
tcgetattr()	3347
tcgetpgrp()	3349
tcgetsid()	3351
tcgetsize()	3353
tcinject()	3355

tcischars()	3357
TCP	3359
tcsendbreak()	3362
tcsetattr()	3364
tcsetpgrp()	3367
tcsetsid()	3369
tcsetsize()	3371
tell(), tell64()	3373
telldir()	3375
tempnam()	3377
termios	3379
thread_pool_control()	3385
thread_pool_create()	3387
thread_pool_destroy()	3393
thread_pool_limits()	3395
thread_pool_start()	3397
ThreadCancel(), ThreadCancel_r()	3399
ThreadCreate(), ThreadCreate_r()	3402
ThreadCtl(), ThreadCtl_r()	3408
ThreadDestroy(), ThreadDestroy_r()	3417
ThreadDetach(), ThreadDetach_r()	3420
ThreadJoin(), ThreadJoin_r()	3422
time()	3425
timer_create()	3427
timer_delete()	3431
timer_getexpstatus()	3433
timer_getoverrun()	3435
timer_gettime()	3437
timer_settime()	3439
timer_timeout(), timer_timeout_r()	3442
TimerAlarm(), TimerAlarm_r()	3448
TimerCreate(), TimerCreate_r()	3451
TimerDestroy(), TimerDestroy_r()	3454
TimerInfo(), TimerInfo_r()	3456
TimerSettime(), TimerSettime_r()	3460
TimerTimeout(), TimerTimeout_r()	3464
times()	3471
timespec	3473
timespec2nsec()	3474
timezone	3475
tm	3476
tmpfile(), tmpfile64()	3478
tmpnam()	3480
tolower()	3482
toupper()	3484

towctrans()	3486
towlower()	3488
towupper()	3490
trace_func_enter()	3492
trace_func_exit()	3494
trace_here()	3496
trace_logb()	3498
trace_logbc()	3500
trace_logf()	3502
trace_logi()	3504
trace_nlogf()	3506
trace_vnlogf()	3508
TraceEvent()	3510
traceparser()	3524
traceparser_cs()	3526
traceparser_cs_range()	3529
traceparser_debug()	3531
traceparser_destroy()	3533
traceparser_get_info()	3534
traceparser_init()	3537
truncate(), truncate64()	3539
ttyname()	3542
ttyname_r()	3544
tzname	3546
tzset()	3547
 Chapter 23: U	 3549
ualarm()	3550
UDP	3553
ultoa(), ulltoa()	3555
umask()	3557
umount()	3559
UNALIGNED_PUT16()	3561
UNALIGNED_PUT32()	3563
UNALIGNED_PUT64()	3565
UNALIGNED_RET16()	3567
UNALIGNED_RET32()	3568
UNALIGNED_RET64()	3569
uname()	3571
ungetc()	3573
ungetwc()	3575
UNIX	3577
unlink()	3579
unlockpt()	3582

unsetenv()	3584
usleep()	3586
utime()	3588
utimes()	3591
utmp	3594
utmpname()	3596
utoa()	3598
Chapter 24: V	3601
va_arg()	3602
va_copy()	3605
va_end()	3607
va_start()	3608
valloc()	3610
verr(), verrx()	3612
vfork()	3614
vfprintf()	3616
vscanf()	3618
vfwprintf()	3620
vfwscanf()	3622
vprintf()	3624
vscanf()	3626
vslog2f()	3628
vslogf()	3630
vsnprintf()	3632
vsprintf()	3635
vsscanf()	3637
vswprintf()	3639
vswscanf()	3641
vsyslog()	3643
vwarn(), vwarnx()	3645
vwprintf()	3647
vwscanf()	3649
Chapter 25: W	3651
wait()	3652
wait3()	3655
wait4()	3658
waitfor()	3661
waitid()	3663
waitpid()	3666
warn(), warnx()	3669
wcrtomb()	3671
wcscat()	3673

wcschr()	3674
wscmp()	3676
wscoll()	3678
wscpy()	3680
wscspn()	3682
wcsftime()	3684
wcslen()	3686
wcsncat()	3687
wcsncmp()	3689
wcsncpy()	3691
wcspbrk()	3693
wcsrchr()	3695
wcsrtombs()	3697
wcsspn()	3699
wcsstr()	3701
wctod(), wctof(), wctold()	3703
wctoimax(), wctoumax()	3705
wctok()	3707
wctol(), wctoll()	3709
wctombs()	3711
wctoul(), wctoull()	3713
wcsxfrm()	3715
wctob()	3717
wctomb()	3718
wctrans()	3720
wctype()	3722
wmemchr()	3724
wmemcmp()	3726
wmemcpy()	3728
wmemmove()	3730
wmemset()	3732
wordexp()	3734
wordfree()	3735
wprintf()	3736
write()	3737
writeblock()	3742
writew()	3745
wscanf()	3748
 Chapter 26: X	 3751
 Chapter 27: Y	 3753
y0(), y0f()	3754
y1(), y1f()	3756

yn(), ynf()	3758
Chapter 28: Z	3761
Appendix A: SOCKS — A Basic Firewall	3763
About SOCKS	3764
How to SOCKSify a client	3765
What SOCKS expects	3766
Appendix B: Full Safety Information	3767
A	3768
B	3771
C	3772
D	3775
E	3777
F	3779
G	3783
H	3787
I	3788
J	3795
K	3796
L	3797
M	3799
N	3804
O	3806
P	3807
Q	3816
R	3817
S	3821
T	3831
U	3835
V	3836
W	3838
Y	3840
Appendix C: What's New in this Reference?	3841
What's new in QNX Neutrino 6.6?	3842
New entries	3842
Deprecated content	3849
Changed content	3850
Errata	3867
What's new in QNX Neutrino 6.5.0 Service Pack 1?	3873
New entries	3873

Changed content	3873
Errata	3876
What's new in QNX Neutrino 6.5.0?	3881
New entries	3881
Changed content	3885
Errata	3897
What's new in QNX Neutrino 6.4.1?	3901
New entries	3901
Deprecated content	3903
Changed content	3903
Errata	3908
What's new in QNX Neutrino 6.4.0?	3910
New entries	3910
Deprecated content	3913
Changed content	3914
Errata	3917
What's new in QNX Neutrino 6.3.2?	3922
Errata	3922
What's new in the QNX Neutrino Core OS 6.3.2?	3923
New entries	3923
Changed content	3923
What's new in QNX Neutrino 6.3.0 Service Pack 2?	3925
New entries	3925
Changed content	3925
Errata	3925
What's new in QNX Neutrino 6.3.0 Service Pack 1?	3927
New entries	3927
Changed content	3927
Errata	3928
What's new in QNX Neutrino 6.3.0?	3930
New entries	3930
Deprecated content	3932
What's new in QNX Neutrino 6.2.1?	3934
New entries	3934
Changed content	3934
Errata	3934
What's new in QNX Neutrino 6.2?	3938
New entries	3938
Deprecated content	3941
Errata	3941
What's new in QNX Neutrino 6.1.0?	3942
New entries	3942
Deprecated content	3943
Glossary	3945

About This Reference

The *C Library Reference* describes the C functions, data types, and protocols that are included as part of the QNX Neutrino RTOS.

For information about:	See:
Specific entries	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
An overview of what you'll find in each entry	What's in a Function Description? (p. 45)
Descriptions of the manifests that you can use for compile-time changes or inspection	Manifests (p. 51)
The SOCKS package	SOCKS — A Basic Firewall
Complete safety information	Full Safety Information
Changes to this reference	What's New in this Reference?
Terms used in QNX Neutrino documentation	Glossary

For information about programming in QNX Neutrino, see *Get Programming with the QNX Neutrino RTOS* and the *QNX Neutrino Programmer's Guide*.

For information about licensing, see:

- the QNX Development Suite License Guide at <http://licensing.qnx.com/license-guide/>
- the Third Party License Terms List at <http://licensing.qnx.com/third-party-terms/>
- the matrix of all the versions of all our legal documents at <http://licensing.qnx.com/document-archive/>

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	<code>if(stream == NULL)</code>
Command options	<code>-lR</code>
Commands	<code>make</code>
Environment variables	<i>PATH</i>
File and pathnames	<code>/dev/null</code>
Function names	<code>exit()</code>
Keyboard chords	Ctrl –Alt –Delete
Keyboard input	Username
Keyboard keys	Enter
Program output	login:
Variable names	<i>stdin</i>
Parameters	<i>parm1</i>
User-interface components	Navigator
Window title	Options

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective → Show View** .

We use notes, cautions, and warnings to highlight important messages:



Notes point out something important or useful.



Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.



Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (www.qnx.com). You'll find a wide range of support options, including community forums.

Chapter 1

What's in a Function Description?

An overview of what's in the documentation for a function

The entry for each function in this reference typically includes the following sections:

- [Synopsis](#) (p. 45)
- [Arguments](#) (p. 45)
- [Library](#) (p. 45)
- [Description](#) (p. 46)
- [Returns](#) (p. 46)
- [Errors](#) (p. 46)
- [Examples](#) (p. 46)
- [Classification](#) (p. 46)
- [Function safety](#) (p. 49)
- [See also](#) (p. 50)

Synopsis:

This section gives the header files that should be included within a source file that references the function or macro. It also shows an appropriate declaration for the function or for a function that could be substituted for a macro. This declaration isn't included in your program; only the header file(s) should be included.

When a pointer argument is passed to a function that doesn't modify the item indicated by that pointer, the argument is shown with `const` before the argument. For example, the following indicates that the array pointed at by *string* isn't changed:

```
const char *string
```

Arguments:

This section gives a brief description of the arguments to the function.

Library:

The section indicates the library that you need to bind with your application in order to use the function.

To link against a library, use the `-l` option to `qcc`, omitting the `lib` prefix and any extension from the library's name. For example, to link against `libsocket`, specify `-l socket`. For more information, see the Compiling and Debugging chapter of the QNX Neutrino *Programmer's Guide*.

Description:

This section describes the function or macro.

Returns:

This section gives the return value (if any) for the function or macro.

Errors:

This section describes the special values that the function might assign to the global variable `errno` (p. 539).



This section doesn't necessarily list *all* of the values that the function could set `errno` to.

Examples:

This optional section gives one or more examples of the use of the function. The examples are often just code snippets, not complete programs.

Classification:

This section tells where the function or macro is commonly found, which may be helpful when porting code from one environment to another. Here are the classes:

ANSI

These functions or macros are defined by the ANSI C99 standard.

Large-file support

These functions support 64-bit offsets.

NetBSD

These functions are part of the NetBSD free, open-source system. For more information, see <http://netbsd.org/>.

POSIX 1003.1

These functions are specified in the document *Information technology — Portable Operating System Interface* (IEEE Std 1003.1, 2004 Edition).

This standard incorporates the POSIX 1003.2-1992 and 1003.1-1996 standards, the approved drafts (POSIX 1003.1a, POSIX 1003.1d, POSIX 1003.1g and POSIX 1003.1j) and the Standard Unix specification. A joint technical working group — the Austin Common Standards Revision Group (CSRG) — was formed to merge these standards.



For information about the many POSIX drafts and standards, see the IEEE website at <http://www.ieee.org/>.

A classification of “POSIX 1003.1” can be followed by one or more codes that indicate which option or options the functions belong to. The codes include the following:

Code	Meaning
ADV	Advisory Information
AIO	Asynchronous Input/Output
BAR	Barriers
CPT	Process CPU-Time Clocks
CS	Clock Selection
CX	Extension to the ISO C standard
FSC	File Synchronization
MF	Memory Mapped Files
ML	Process Memory Locking
MLR	Range Memory Locking
MPR	Memory Protection
MSG	Message Passing
OB	Obsolescent
PS	Process Scheduling
RTS	Realtime Signals Extension
SEM	Semaphores
SHM	Shared Memory Objects
SIO	Synchronous Input/Output
SPI	Spin Locks
SPN	Spawn
TCT	Thread CPU-Time Clocks
THR	Threads
TMO	Timeouts
TMR	Timers

Code	Meaning
TPI	Thread Priority Inheritance
TPP	Thread Priority Protection
TPS	Thread Execution Scheduling
TSA	Thread Stack Address Attribute
TSF	Thread-Safe Functions
TSH	Thread Process-Shared Synchronization
TSS	Thread Stack Size Attribute
TYM	Typed Memory Objects
XSI	X/Open Systems Interfaces Extension
XSR	XSI Streams

If two codes are separated by a space, you need to use both options; if the codes are separated by a vertical bar (|), the functionality is supported if you use either option.

For more information, see the *Standard for Information Technology — Portable Operating System Interface: Base Definitions*.

QNX 4

These functions or macros are neither ANSI nor POSIX. They perform a function related to the QNX OS version 4. They may be found in other implementations of C for personal computers with the QNX 4 OS. Use these functions with caution if portability is a consideration.



Any QNX 4 functions in the C library are provided *only* to make it easier to port QNX 4 programs. Don't use these in new programs.

QNX Neutrino

These functions or macros are neither ANSI nor POSIX. They perform a function related to the QNX Neutrino family of operating systems, including the QNX Neutrino RTOS and the BlackBerry 10 OS. They may be found in other implementations of C for personal computers with the QNX Neutrino RTOS. Use these functions with caution if portability is a consideration.

RFC 2292

Based on W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998.

SOCKS

These functions are part of the SOCKS package consisting of a proxy server, client programs, and a library (`libsocks`) for adapting other applications into new client programs. For more information, see the appendix [SOCKS — A Basic Firewall](#).

Unix

These Unix-class functions reside on some Unix systems, but are outside of the POSIX or ANSI standards.

We've created the following Unix categories to differentiate:

Legacy Unix

Functions included for backwards compatibility only. New applications shouldn't use these functions.

Unix

Other Unix functions.

Function safety:

This section summarizes whether or not it's safe to use the C library functions in certain situations:

Cancellation point

Indicates whether calling a function may or may not cause the thread to be terminated if a cancellation is pending.

Interrupt handler

An interrupt-safe function behaves as documented even if used in an interrupt handler. Functions flagged as interrupt-unsafe shouldn't be used in interrupt handlers.

Signal handler

A signal-safe function behaves as documented even if called from a signal handler *even if the signal interrupts a signal-unsafe function*.

Some of the signal-safe functions modify `errno` on failure. If you use any of these in a signal handler, asynchronous signals may have the side effect of modifying `errno` in an unpredictable way. If any of the code that can be interrupted checks the value of `errno` (this also applies to library calls, so

you should assume that most library calls may internally check *errno*), make sure that your signal handler saves *errno* on entry and restores it on exit.

All of the above also applies to signal-unsafe functions, with one exception: if a signal handler calls a signal-unsafe function, make sure that signal doesn't interrupt a signal-unsafe function.

Thread

A thread-safe function behaves as documented even if called in a multi-threaded environment. Most functions in the QNX Neutrino libraries are thread-safe.



- The “safety” designations documented in this manual are valid for the current release and could change in future versions.
- It isn't safe to use floating-point operations in Interrupt Service Routines (ISRs) or signal handlers.

For a summary, see the [Full Safety Information](#) appendix.

See also:

This optional section provides a list of related functions or macros as well as pertinent docs to look for more information.

Chapter 2

Manifests

Manifests are used by C/C++ for compile-time changes or inspection. Here are the defined items:

Manifest	Header file to include	Description
__BEGIN_DECLS	sys/platform.h	Denotes start of C code for a C++ compiled program.
__BIGENDIAN__	sys/platform.h	Code is compiled for a big-endian target.
__CHAR_SIGNED__	sys/platform.h	Code is compiled with the char type defaulting to signed.
__CHAR_UNSIGNED__	sys/platform.h	Code is compiled with the char type defaulting to unsigned.
__END_DECLS	sys/platform.h	Denotes end of C code for a C++ compiled program
__INT_BITS__	sys/platform.h	The number of bits in the int datatype.
__LITTLEENDIAN__	sys/platform.h	Code is compiled for a little-endian target.
__LONG_BITS__	sys/platform.h	The number of bits in the long datatype.
_NTO_VERSION	sys/neutrino.h	A version number times 100 (e.g. 2.00 is 200).
__PTR_BITS__	sys/platform.h	The number of bits in a void pointer.
__OPTIMIZE__	sys/platform.h	Code is compiled for optimization.
__QNX__	N/A	The target is for a QNX operating system (QNX 4, QNX Neutrino, or QNX Neutrino).

Manifest	Header file to include	Description
__QNXNTO__	N/A	The target is the QNX Neutrino RTOS or the QNX Neutrino RTOS.

Chapter 3

A

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>APS_INIT_DATA()</i>	SchedCtl() (p. 2745)

This chapter describes the API components whose names begin with “A”.

abort()

Raise the SIGABRT signal to terminate program execution

Synopsis:

```
#include <stdlib.h>

void abort( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `abort()` function causes abnormal process termination to occur by means of the function call [raise\(SIGABRT\)](#) (p. 2482), unless the signal SIGABRT is caught and the signal handler doesn't return. If the signal SIGABRT is caught and the signal handler returns, the signal handler is removed and `raise(SIGABRT)` is called again. Note that prior to calling `raise()`, `abort()` ensures that SIGABRT isn't ignored or blocked.

Returns:

The `abort()` function doesn't return to its caller.

Examples:

```
#include <stdlib.h>

int main( void )
{
    int major_error = 1;

    if( major_error )
        abort();

    /* You'll never get here. */
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

A strictly-conforming POSIX application *can't* assume that the *abort()* function is safe to use in a signal handler on other platforms.

abs()

Return the absolute value of an integer

Synopsis:

```
#include <stdlib.h>

int abs( int j );
```

Arguments:

j

The number you want the absolute value of.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *abs()* function returns the absolute value of the integer argument *j*. The behavior is undefined if the result can't be represented as an `int`. Note that the absolute value of the most negative number can't be represented in two's complement.

Returns:

The absolute value of its argument.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d %d %d\n", abs (-5), abs (0), abs (5));
    return EXIT_SUCCESS;
}
```

produces the following output:

```
5 0 5
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

accept()

Accept a connection on a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept( int s,
            struct sockaddr * addr,
            socklen_t * addrlen );
```

Arguments:

s

A socket that's been created with [socket\(\)](#) (p. 3075).

addr

A result parameter that's filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the connection was made.

addrlen

A value-result parameter. It should initially contain the amount of space pointed to by *addr*; on return it contains the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *accept()* function:

1. Extracts the first connection request on the queue of pending connections.
2. Creates a new socket with the same properties of *s*, where *s* is a socket that's been created with [socket\(\)](#) (p. 3075), bound to an address with [bind\(\)](#) (p. 208), and is listening for connections after a [listen\(\)](#) (p. 1523).
3. Allocates a new file descriptor for the socket.

If no pending connections are present on the queue, and the socket isn't marked as nonblocking, *accept()* blocks the caller until a connection is present. If the socket is marked as nonblocking and no pending connections are present on the queue, *accept()* returns an error as described below. The accepted socket may *not* be used to accept more connections. The original socket *s* remains open.

If you do a *select()* (p. 2791) for read on an unconnected socket (on which a *listen()* has been done), the *select()* indicates when a connect request has occurred. In this way, an *accept()* can be made that won't block. For more information, see *select()* (p. 2791).

For certain protocols that require an explicit confirmation, *accept()* can be thought of as merely dequeuing the next connection request and *not* implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

You can obtain user-connection request data without confirming the connection by:

- Issuing a *recvmsg()* (p. 2539) call with a *msg_iovlen* of 0 and a nonzero *msg_controllen*
- Or
- Issuing a *getsockopt()* (p. 962) request.

Similarly, you can provide user-connection rejection information by issuing a *sendmsg()* (p. 2833) call with only the control information, or by calling *setsockopt()* (p. 2909).

Returns:

A descriptor for the accepted socket, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EAGAIN

Insufficient resources to create the new socket.

EBADF

Invalid descriptor *s*.

EFAULT

The *addr* parameter isn't in a writable part of the user address space.

EINVAL

You called *accept()* on a socket that you hadn't called *listen()* on.

EOPNOTSUPP

The referenced socket isn't a SOCK_STREAM socket.

ESRCH

Can't find the socket manager.

EWOULDBLOCK

The socket is marked nonblocking and no connections are present to be accepted.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

access()

Check to see if a file or directory can be accessed

Synopsis:

```
#include <unistd.h>

int access( const char * path,
            int amode );
```

Arguments:***path***

The path to the file or directory that you want to access.

amode

The access mode you want to check. This must be either:

- F_OK — test for file existence.

or a bitwise ORing of the following access permissions to be checked, as defined in the header `<unistd.h>`:

- R_OK — test for read permission.
- W_OK — test for write permission.
- X_OK — for a directory, test for search permission. Otherwise, test for execute permission.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `access()` function checks to see if the file or directory specified by *path* exists and if it can be accessed with the file access permissions given by *amode*. However, unlike other functions ([open\(\)](#) (p. 1905) for example), it uses the real user ID and real group ID in place of the effective user and group IDs.

Returns:

0

The file or directory exists and can be accessed with the specified mode.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EACCES**

The permissions specified by *amode* are denied, or search permission is denied on a component of the path prefix.

EINVAL

An invalid value was specified for *amode*.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of the *path* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT

A component of the path isn't valid.

ENOSYS

The *access()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix isn't a directory.

EROFS

Write access was requested for a file residing on a read-only file system.

Examples:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    if( argc!= 2 ) {
        fprintf( stderr,
            "use: readable <filename>\n" );
        return EXIT_FAILURE;
    }
}
```

```
if( !access( argv[1], R_OK ) ) {  
    printf( "ok to read %s\n", argv[1] );  
    return EXIT_SUCCESS;  
} else {  
    perror( argv[1] );  
    return EXIT_FAILURE;  
}  
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

acl_add_perm()

Add a permission to an access control list (ACL) permission set

Synopsis:

```
#include <sys/acl.h>

int acl_add_perm( acl_permset_t permset_d,
                  acl_perm_t perm );
```

Arguments:

permset_d

The permission set that you want to modify.

perm

The permission that you want to add; one of the following:

- ACL_EXECUTE
- ACL_READ
- ACL_WRITE

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *acl_add_perm()* function adds a permission to an ACL permission set.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *permset_d* argument isn't a valid descriptor for a permission set, or *perm* doesn't contain a valid permission value.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_calc_mask()

Calculate the group class mask for an access control list (ACL)

Synopsis:

```
#include <sys/acl.h>

int acl_calc_mask( acl_t *acl_p );
```

Arguments:

acl_p

A pointer to the ACL whose mask you want to calculate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_calc_mask()* function calculates the file group mask for an ACL, as the union of the permissions granted by the `ACL_GROUP` (named group), `ACL_GROUP_OBJ` (owning group), and `ACL_USER` (named user) tags. If the ACL already has a mask entry, the function updates it; otherwise it adds a mask entry.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acl_p* argument isn't the address of a pointer to a valid ACL.

ENOMEM

There wasn't enough memory available to allocate space for a new mask entry.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_clear_perms()

Clear all permissions from an ACL permission set

Synopsis:

```
#include <sys/acl.h>

int acl_clear_perms( acl_permset_t permset_d );
```

Arguments:

permset_d

The permissions set that you want to clear.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *acl_clear_perms()* function clears all permissions from an ACL permissions set.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *permset_d* argument isn't a valid descriptor for a permission set within an ACL entry.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

acl_copy_entry()

Copy the contents of one ACL entry into another

Synopsis:

```
#include <sys/acl.h>

int acl_copy_entry( acl_entry_t dest_d,
                   acl_entry_t src_d );
```

Arguments:

dest_d

The descriptor for the destination entry.

src_d

The descriptor for the entry that you want to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_copy_entry()* function copies the contents of the ACL entry referred to by *src_d* into the entry referred to by *dest_d*.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

Either *src_d* or *dest_d* isn't a valid descriptor for an ACL entry, or they reference the same ACL entry.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_copy_ext()

Copy an ACL from system space to user space

Synopsis:

```
#include <sys/acl.h>

ssize_t acl_copy_ext( void *buf_p,
                     acl_t acl,
                     ssize_t size );
```

Arguments:

buf_p

A pointer to the buffer that you want to copy the ACL into.

acl

The ACL that you want to copy.

size

The size of the buffer pointed to by *buf_p*, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_copy_ext()* function copies an access control list from system-managed space to the user-managed buffer pointed to by *buf_p*. You can call [acl_size\(\)](#) (p. 112) to determine the size of buffer needed.

Returns:

The number of bytes placed in the buffer, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *size* argument is smaller than the size of the contiguous, persistent form of the ACL, *acl* doesn't point to a valid ACL, or the ACL contains some improperly formed entries.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_copy_int()

Copy an ACL from user space to system space

Synopsis:

```
#include <sys/acl.h>

acl_t acl_copy_int( const void *buf_p );
```

Arguments:

buf_p

A pointer to a buffer that contains an exportable, contiguous, persistent form of the ACL that you want to copy.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *acl_copy_int()* function copies an access control list from user-managed space to system-managed space. When you're finished with the ACL, you should call [*acl_free\(\)*](#) (p. 84) to release it.

Returns:

A pointer to the ACL in working storage, or NULL if an error occurred ([*errno*](#) (p. 539) is set).

Errors:

EINVAL

The buffer doesn't contain a valid external-form ACL.

ENOMEM

There wasn't enough memory to create the ACL in working storage.

ENOTSUP

The ACL type isn't `ACL_TYPE_ACCESS` or `ACL_TYPE_DEFAULT`.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_create_entry()

Create an entry in an access control list

Synopsis:

```
#include <sys/acl.h>

int acl_create_entry( acl_t *acl_p,
                    acl_entry_t *entry_p );
```

Arguments:

acl_p

The address of a pointer to the ACL that you want to add an entry to. If *acl_create_entry()* needs to reallocate the entry, it updates this argument.

entry_p

A pointer to a location where the function can store a pointer to the new entry.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_create_entry()* function creates a new entry in an access control list. The new entry is initialized as follows:

- The tag type is set to `ACL_UNDEFINED_TAG`.
- The qualifier is set to `ACL_UNDEFINED_ID`.
- No permissions are enabled.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *acl_p* argument doesn't point to a valid ACL.

ENOMEM

There wasn't enough memory to allocate the new entry.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_delete_entry()

Delete an entry from an access control list (ACL)

Synopsis:

```
#include <sys/acl.h>

int acl_delete_entry( acl_t acl,
                    acl_entry_t entry_d );
```

Arguments:

acl

A pointer to the ACL that you want to delete an entry from.

entry_d

The descriptor of the entry that you want to delete.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_delete_entry()* function deletes an entry from an access control list.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acl* argument doesn't point to a valid ACL, or *entry_d* isn't a valid descriptor for an entry in *acl*.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_delete_perm()

Delete a permission from an ACL permissions set

Synopsis:

```
#include <sys/acl.h>

int acl_delete_perm( acl_permset_t permset_d,
                    acl_perm_t perm );
```

Arguments:

permset_d

The permission set that you want to modify.

perm

The permission that you want to delete; one of the following:

- ACL_EXECUTE
- ACL_READ
- ACL_WRITE

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *acl_delete_perm()* function deletes the given permission from an ACL permission set. Attempting to delete a permission that isn't in the set isn't considered to be an error.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *permset_d* argument isn't a valid descriptor for an ACL permission set, or *perm* isn't a valid permission value.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_dup()

Duplicate an access control list (ACL) in a working storage area

Synopsis:

```
#include <sys/acl.h>
acl_t acl_dup( acl_t acl );
```

Arguments:

acl

A pointer to the ACL that you want to duplicate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_dup()* function duplicates an access control list in a working storage area. When you no longer need the new ACL, call [acl_free\(\)](#) (p. 84) to release it.

Returns:

A pointer to the copy, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acl* argument doesn't point to a valid ACL.

ENOMEM

Not enough memory.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_free()

Free the working storage area allocated for an access control list (ACL) data object

Synopsis:

```
#include <sys/acl.h>

int acl_free( void *obj_p );
```

Arguments:

obj_p

A pointer to the object you want to release. This can be a pointer to an ACL, an ACL qualifier, or a string allocated by one of the ACL functions.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *acl_free()* function releases a work storage area that was allocated for an ACL data object.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of the *obj* argument is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_from_text()

Create an access control list (ACL) from text

Synopsis:

```
#include <sys/acl.h>

acl_t acl_from_text( const char *buf_p );
```

Arguments:

buf_p

A pointer to a buffer that contains the text form of the ACL that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_from_text()* function converts an access control list from text form into the internal form of an ACL. This function accepts the long and short text forms of an ACL:

tag_type:qualifier:permissions

In the long form, *tag_type* is *user*, *group*, *other*, or *mask*, the *qualifier* is the name or numeric ID of a user or group, and the *permissions* are in the form *rwX*, with a hyphen (-) replacing any permissions that aren't granted.

In the short form, you can abbreviate the *tag_type* to its first letter, and the permissions can contain at most one each of *r*, *w*, and *x*, in any order.

When you're finished with the resulting ACL, you should call [acl_free\(\)](#) (p. 84) to release it.

Returns:

A pointer to the internal representation of the ACL, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The contents of the buffer couldn't be converted into an ACL.

ENOMEM

There wasn't enough memory available to allocate for the ACL in working storage.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_get_entry()

Get an entry in an access control list (ACL)

Synopsis:

```
#include <sys/acl.h>

int acl_get_entry( acl_t acl,
                  int entry_id,
                  acl_entry_t *entry_p );
```

Arguments:

acl

A point to the access control list that you want to get an entry from.

entry_id

The ID of the entry you want to get; one of the following:

- ACL_FIRST_ENTRY — get the first entry in the ACL.
- ACL_NEXT_ENTRY — get the next entry in the ACL.

entry_p

A pointer to a location where the function can store a pointer to the entry.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_get_entry()* function gets an entry from an access control list. Each ACL remembers which entry is the current one.

Returns:

1

An entry was found.

0

The ACL has no entries, or the last entry was returned by a previous call to `acl_get_entry()`.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The `acl` argument doesn't point to a valid ACL, or `entry_id` isn't `ACL_FIRST_ENTRY` or `ACL_NEXT_ENTRY`.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_get_fd()

Get the access control list associated with a file descriptor

Synopsis:

```
#include <sys/acl.h>

acl_t acl_get_fd( int fd );
```

Arguments:

fd

A file descriptor associated with the file whose ACL you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `acl_get_fd()` function gets the ACL of the object associated with the given file descriptor, puts it into working storage, and returns a pointer to that storage. When you're finished with the ACL, you should call [acl_free\(\)](#) (p. 84) to release it.



The ACL in working storage is independent of the file or directory's ACL. Changes that you make to the copy in working storage don't affect the file or directory's ACL.

Returns:

A pointer to the ACL in working storage, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The *fd* argument isn't a valid file descriptor.

ENOMEM

There wasn't enough memory available to create the ACL in working storage.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_get_file()

Get the ACL for a given path

Synopsis:

```
#include <sys/acl.h>

acl_t acl_get_file( const char *path_p,
                   acl_type_t type );
```

Arguments:

path_p

The path that you want to get the ACL for.

type

The type of ACL; this must currently be ACL_TYPE_ACCESS.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_get_file()* function gets the ACL associated with the given file or directory, copies it into working storage, and returns a pointer to that storage. When you're finished with the ACL, you should call [acl_free\(\)](#) (p. 84) to release it.



The ACL in working storage is independent of the file or directory's ACL. Changes that you make to the copy in working storage don't affect the file or directory's ACL.

Returns:

A pointer to the ACL in working storage, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission was denied for a component of the path prefix, or the object exists and the process doesn't have the appropriate access rights.

EINVAL

The *type* argument isn't ACL_TYPE_ACCESS.

ENAMETOOLONG

The length of the *path_p* argument exceeds PATH_MAX.

ENOENT

The named object doesn't exist, or *path_p* is an empty string.

ENOMEM

There wasn't enough memory available to create the ACL in working storage.

ENOTDIR

A component of the path prefix isn't a directory.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_get_permset()

Get a permission set from an ACL entry

Synopsis:

```
#include <sys/acl.h>

int acl_get_permset( acl_entry_t entry_d,
                    acl_permset_t *permset_p );
```

Arguments:

entry_d

A descriptor of the entry that you want to get the permissions from.

permset_p

A pointer to a location where the function can store the permission set.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_get_permset()* function gets the permissions set from an ACL entry.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *entry_d* argument isn't a valid descriptor for an ACL entry.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_get_qualifier()

Get the qualifier from an ACL entry

Synopsis:

```
#include <sys/acl.h>

void *acl_get_qualifier( acl_entry_t entry_d );
```

Arguments:

entry_d

The entry that you want to get the qualifier from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_get_qualifier()* function gets the qualifier from an ACL entry and returns a pointer to a copy of it. The data type of the pointer depends on the type of the entry:

Entry type	Pointer type
ACL_USER	uid_t *
ACL_GROUP	gid_t *
Other types	(void *)NULL, and the function fails



The `uid_t` and `gid_t` data types are defined in `<sys/types.h>`.

When you're finished with the copy of the qualifier, use [acl_free\(\)](#) (p. 84) to release it.

Returns:

A pointer to a copy of the qualifier, or `NULL` if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *entry_d* argument isn't a valid descriptor for an ACL entry, or the value of the tag type in the entry isn't ACL_USER or ACL_GROUP.

ENOMEM

There wasn't enough memory to create a copy of the qualifier.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_get_tag_type()

Get the type of tag from an ACL entry

Synopsis:

```
#include <sys/acl.h>

int acl_get_tag_type( acl_entry_t entry_d,
                    acl_tag_t *tag_type_p );
```

Arguments:

entry_d

The descriptor of the entry that you want to get the tag type of.

tag_type_p

A pointer to a location where the function can store the tag type; one of the following:

- ACL_GROUP — a named group.
- ACL_GROUP_OBJ — the owning group.
- ACL_MASK — the maximum permissions allowed for named users, named groups, and the owning group.
- ACL_OTHER — users whose process attributes don't match any other ACL entry; the “world”.
- ACL_USER — named users.
- ACL_USER_OBJ — the owning user.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_get_tag_type()* function gets the tag type from an ACL entry and stores it in the location pointed to by *tag_type_p*.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set, and the location pointed to by *tag_type_p* isn't changed).

Errors:**EINVAL**

The *entry_d* argument isn't a valid descriptor for an ACL entry.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_init()

Allocate and initialize an access control list (ACL) working storage area

Synopsis:

```
#include <sys/acl.h>

acl_t acl_init( int count );
```

Arguments:

count

The number of ACL entries that you want to allocate and initialize.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_init()* function allocates and initializes working storage for an ACL of at least *count* entries.

Returns:

A pointer to the allocated area, or `NULL` if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *count* argument is less than zero.

ENOMEM

Not enough memory.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_set_fd()

Set the access ACL for the object associated with a file descriptor

Synopsis:

```
#include <sys/acl.h>

int acl_set_fd( int fd,
               acl_t acl );
```

Arguments:

fd

A file descriptor associated with the object whose ACL you want to set.

acl

The ACL that you want to assign to the object.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_set_fd()* function sets the access ACL for an object associated with a file descriptor. The calling process's effective user ID must match the owner of the directory, or the process must have appropriate privileges in order to set the ACL. The ACL must also be valid, as defined by *acl_valid()* (p. 116).

If this function succeeds, the object's file permission bits might change; if it fails, the object's ACL and file permission bits aren't changed.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EBADF

The *fd* argument isn't a valid file descriptor.

EINVAL

The *acl* argument doesn't point to a valid ACL.

ENOSPC

The directory or filesystem that would contain the new ACL can't be extended, or the filesystem is out of file allocation resources.

EPERM

The process doesn't have the appropriate privileges to set the ACL.

EROFS

The filesystem that the object is on is currently read-only.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_set_file()

Set the access control list for a path

Synopsis:

```
#include <sys/acl.h>

int acl_set_file( const char *path_p,
                  acl_type_t type,
                  acl_t acl );
```

Arguments:

path_p

The path that you want to set the ACL for.

type

The type of ACL; this must currently be ACL_TYPE_ACCESS.

acl

The ACL that you want to assign to the object.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_set_file()* function sets the access control list for a given path. The calling process's effective user ID must match the owner of the directory, or the process must have appropriate privileges in order to set the ACL. The ACL must also be valid, as defined by *acl_valid()* (p. 116).

If this function succeeds, the object's file permission bits might change; if it fails, the object's ACL and file permission bits aren't changed.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EACCES

Search permission was denied for a component of the path prefix, or the object exists and the process doesn't have the appropriate access rights.

EINVAL

The *acl* argument doesn't point to a valid ACL, or the *type* argument isn't `ACL_TYPE_ACCESS`.

ENAMETOOLONG

The length of the *path_p* argument exceeds `PATH_MAX`.

ENOENT

The named object doesn't exist, or *path_p* is an empty string.

ENOSPC

The directory or filesystem that would contain the new ACL can't be extended, or the filesystem is out of file allocation resources.

ENOTDIR

A component of the path prefix isn't a directory.

EPERM

The process doesn't have the appropriate privileges to set the ACL.

EROFS

The filesystem that the object is on is currently read-only.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_set_permset()

Set the permissions set in an ACL entry

Synopsis:

```
#include <sys/acl.h>

int acl_set_permset( acl_entry_t entry_d,
                    acl_permset_t permset_p );
```

Arguments:

entry_d

A descriptor of the entry that you want to set the permissions from.

permset_p

The permissions set that you want to store in the entry.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_set_permset()* function stores the given permissions set in an ACL entry.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *entry_d* argument isn't a valid descriptor for an ACL entry, the *permset_p* isn't a valid descriptor for a permission set within an ACL entry, or the permissions include invalid values.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_set_qualifier()

Set the qualifier for an ACL entry

Synopsis:

```
#include <sys/acl.h>

int acl_set_qualifier( acl_entry_t entry_d,
                      const void *tag_qualifier_p );
```

Arguments:

entry_d

The descriptor of the entry whose qualifier you want to set.

tag_qualifier_p

A pointer to the qualifier (see below).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *acl_set_qualifier()* function sets the qualifier in an ACL entry. The data type of the value referred to by the *tag_qualifier_p* argument depends on the entry type:

Entry type	Data type
ACL_USER	uid_t
ACL_GROUP	gid_t
Other types	Invalid; the function fails



The uid_t and gid_t data types are defined in <sys/types.h>.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:**EINVAL**

The *entry_d* argument isn't a valid descriptor for an ACL entry, the value of the tag type in the entry isn't ACL_USER or ACL_GROUP, or the value pointed to by *tag_qualifier_p* isn't valid.

ENOMEM

There wasn't enough memory to create a copy of the qualifier.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_set_tag_type()

Set the tag type of an ACL entry

Synopsis:

```
#include <sys/acl.h>

int acl_set_tag_type( acl_entry_t entry_d,
                     acl_tag_t tag_type );
```

Arguments:

entry_d

The descriptor of the ACL entry whose type you want to set.

tag_type

The type that you want to assign to the entry; one of the following:

- ACL_GROUP — a named group.
- ACL_GROUP_OBJ — the owning group.
- ACL_MASK — the maximum permissions allowed for named users, named groups, and the owning group.
- ACL_OTHER — users whose process attributes don't match any other ACL entry; the “world”.
- ACL_USER — named users.
- ACL_USER_OBJ — the owning user.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *acl_set_tag_type()* function sets the type of an ACL entry.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *entry_d* argument isn't a valid descriptor for an ACL entry.

ENOTSUP

The *tag_type* isn't valid.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_size()

Determine the size of the external form of an ACL

Synopsis:

```
#include <sys/acl.h>
ssize_t acl_size( acl_t acl );
```

Arguments:

acl
A pointer to an ACL.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *acl_size()* function calculates the size, in bytes, of the buffer required to hold the exportable, contiguous, persistent form of the given ACL.

Returns:

The size, in bytes, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EINVAL
The *acl* argument doesn't point to a valid ACL.

Classification:

QNX Neutrino (p. 46)
This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

acl_to_text()

Convert an access control list (ACL) into text

Synopsis:

```
#include <sys/acl.h>

char *acl_to_text( acl_t acl,
                  ssize_t *len_p );
```

Arguments:

acl

A pointer to the ACL that you want to convert into text.

len_p

NULL, or a pointer to a location where the function can store the length of the string (not including the terminating null character).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *acl_to_text()* function converts an access control list into the long text form:

tag_type:qualifier:permissions

The *tag_type* is *user*, *group*, *other*, or *mask*, the *qualifier* is the name or numeric ID of a user or group, and the *permissions* are in the form *rwX*, with a hyphen (-) replacing any permissions that aren't granted.

When you're finished with the text form of the ACL, you should call [acl_free\(\)](#) (p. 84) to release it.

Returns:

A pointer to the text form of the ACL, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acl* argument doesn't point to a valid ACL, or the ACL contains some improperly formed entries or otherwise can't be converted.

ENOMEM

There wasn't enough memory available to allocate for the string.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acl_valid()

Validate an access control list (ACL)

Synopsis:

```
#include <sys/acl.h>

int acl_valid( acl_t acl );
```

Arguments:

acl

A pointer to the ACL that you want to check.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *acl_valid()* function checks an access control list to make sure it's valid:

- The ACL must have one each of the `ACL_USER_OBJ`, `ACL_GROUP_OBJ`, and `ACL_OTHER` entries.
- If the ACL has any `ACL_USER` (named user) or `ACL_GROUP` (named group) entries, there must be one `ACL_MASK` entry.
- There must not be more than one entry for the same named user or group.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acl* argument doesn't point to a valid ACL, the ACL is missing some of the required entries, or there are entries that aren't unique.

ENOTSUP

The ACL type isn't `ACL_TYPE_ACCESS` or `ACL_TYPE_DEFAULT`.

Classification:

[QNX Neutrino](#) (p. 46)

This function is based on the withdrawn POSIX draft P1003.1e.

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acos(), acosf(), acosl()

Compute the arccosine of an angle

Synopsis:

```
#include <math.h>

double acos( double x );
float acosf( float x );
long double acosl( long double x );
```

Arguments:

x

The cosine for which you want to find the angle.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the arccosine (specified in radians) of *x*.

Returns:

The arccosine in the range $(0, \pi)$. For finite values not in the range $[-1,1]$, these functions return NaN. The return value for $\pm\text{Inf}$ is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", acos(.5) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
1.047197
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

acosh(), acoshf(), acoshl()

Compute the inverse hyperbolic cosine

Synopsis:

```
#include <math.h>

double acosh( double x );
float acoshf( float x );
long double acoshl( long double x );
```

Arguments:

x

The value for which you want to compute the inverse hyperbolic cosine.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the inverse hyperbolic cosine (specified in radians) of *x*.

Returns:

The inverse hyperbolic cosine of *x* (specified in radians). For finite values of $x < 1$, these functions return NaN. The return value when *x* is `-Inf` is NaN.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", acosh( 1.5 ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0.962424
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

addrinfo

TCP/IP address information

Synopsis:

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    char * ai_canonname;
    struct sockaddr * ai_addr;
    struct addrinfo * ai_next
};
```

Description:

The `addrinfo` structure describes address information for use with TCP/IP. To get this information, call [`getaddrinfo\(\)`](#) (p. 818); to free a linked list of these structures, call [`freeaddrinfo\(\)`](#) (p. 719).

The `addrinfo` structure includes these members:

ai_flags

Flags. Includes `AI_PASSIVE`, `AI_CANONNAME`, and `AI_NUMERICHOST`. For a complete list, see `<netdb.h>`.

ai_family

Protocol family. Includes `PF_UNSPEC` and `PF_INET`. For a complete list, see `<sys/socket.h>`.

ai_socktype

Socket type. Includes `SOCK_STREAM` and `SOCK_DGRAM`. For a complete list, see `<sys/socket.h>`.

ai_protocol

Protocol. Includes `IPPROTO_TCP` and `IPPROTO_UDP`. For a complete list, see `<netinet/in.h>`.

ai_addrlen

The length of the `ai_addr` member.

ai_canonname

The canonical name for *nodename*.

ai_addr

Binary socket address.

ai_next

A pointer to the next `addrinfo` structure in the linked list.

Classification:

[*POSIX 1003.1*](#) (p. 46)

aio_cancel()

Cancel an asynchronous I/O operation

Synopsis:

```
#include <aio.h>

int aio_cancel( int fd,
               struct aiocb * aiocbptr );
```

Arguments:

fd

The file descriptor for which you want to cancel asynchronous I/O requests.

aiocbptr

A pointer to an asynchronous I/O control block of type [aiocb](#) (p. 141) for the request you want to cancel, or `NULL` if you want to cancel all requests against the file descriptor.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *aio_cancel()* function attempts to cancel one or more asynchronous I/O requests currently outstanding against a file descriptor.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests that can't be canceled, then the normal asynchronous completion process takes place for those requests when they're completed.

The error status that's associated with requested operations that are successfully canceled is set to `ECANCELED`, and the return status is `-1`. The *aio_cancel()* function doesn't modify the `aiocb` structures for requested operations that aren't successfully canceled.

If *aiocbptr* isn't `NULL`, *aio_cancel()* ignores the *fd* argument and attempts to cancel the I/O operation specified by the `aiocb` control block. The operation isn't cancelled if it's already in progress.

Returns:**AIO_CANCELED**

The requested operation(s) were canceled.

AIO_NOTCANCELED

At least one of the requested operations couldn't be canceled because it was in progress.



A return value of **AIO_NOTCANCELED** doesn't indicate the state of any other operations referenced in the call to *aio_cancel()*. To determine their status, use *aio_error()* (p. 127).

AIO_ALLDONE

All of the operations have already been completed.

-1

An error occurred; *errno* (p. 539) is set.

Errors:**EBADF**

The *filesdes* argument isn't a valid file descriptor.

EINVAL

The control block that *aiocbptr* points to isn't valid (i.e. it hasn't yet been used in any call to *aio_read()* (p. 131) or *aio_write()* (p. 138)).

Classification:

POSIX 1003.1 AIO (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an *ai*_* function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aio_error()

Get the error status for an asynchronous I/O operation

Synopsis:

```
#include <aio.h>

int aio_error( const struct aiocb * aiocbptr );
```

Arguments:

aiocbptr

A pointer to an asynchronous I/O control block of type [aiocb](#) (p. 141) that you want the error status for.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *aio_error()* function returns the error status associated with the `aiocb` structure referenced by the *aiocbptr* argument. The error status for an asynchronous I/O operation is the *errno* value that's set by the corresponding [read\(\)](#) (p. 2502), [write\(\)](#) (p. 3737), or [fsync\(\)](#) (p. 787) operation. If the operation hasn't yet been completed, the error status is `EINPROGRESS`.

Returns:

One of:

- 0 if the operation was completed successfully
- the error status set by *read()*, *write()*, or *fsync()* if the operation wasn't completed successfully
- `EINPROGRESS` if the operation hasn't yet been completed

Errors:

EINVAL

The *aiocbptr* argument doesn't refer to an asynchronous operation whose return status hasn't yet been retrieved.

Classification:

[*POSIX 1003.1 AIO*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an *aio_** function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aio_fsync()

Asynchronously synchronize a file

Synopsis:

```
#include <aio.h>

int aio_fsync( int op,
               struct aiocb * aiocbptr );
```

Arguments:

op

One of the following:

- **O_DSYNC** — complete all queued operations as if done by calling [fdatsync\(\)](#) (p. 631) (i.e. as defined for synchronized I/O data integrity completion).
- **O_SYNC** — complete all queued operations as if done by calling [fsync\(\)](#) (p. 787) (i.e. as defined for synchronized I/O file integrity completion).

aiocbptr

A pointer to an asynchronous I/O control block of type [aiocb](#) (p. 141) for which you want to asynchronously force all queued I/O operations to the synchronized I/O completion state.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *aio_fsync()* function asynchronously forces all I/O operations associated with the file indicated by the file descriptor to the synchronized I/O completion state.

If the function succeeds, only the I/O operations that were queued at the time of the call are guaranteed to have been completed. The completion of subsequent I/O on the file descriptor isn't guaranteed to be completed in a synchronized fashion. If the operation that *aio_fsync()* queues fails, outstanding I/O operations aren't guaranteed to have been completed.

You can pass the *aiocbptr* argument to [aio_error\(\)](#) (p. 127) or [aio_return\(\)](#) (p. 134) to determine the error status and return status for the asynchronous operation.

Returns:

0 if the I/O operation was successfully queued, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The requested asynchronous operation wasn't queued due to temporary resource limitations.

EBADF

The *aio_fildes* member of the *aio_cb* structure referenced by the *aio_cbptr* argument isn't a valid file descriptor that's open for writing.

EINVAL

Synchronized I/O isn't supported for this file.

EINVAL

The *op* argument was something other than `O_DSYNC` or `O_SYNC`.

If any of the queued I/O operations failed, `aio_fsync()` returns the error condition defined for [read\(\)](#) (p. 2502) and [write\(\)](#) (p. 3737). The error is returned in the error status for the asynchronous [fsync\(\)](#) (p. 787) operation, which you can retrieve by using [aio_error\(\)](#) (p. 127).

Classification:

[POSIX 1003.1 AIO](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an *aio_** function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aio_read(), aio_read64()

Asynchronously read from a file

Synopsis:

```
#include <aio.h>

int aio_read( struct aiocb * const aiocbptr );
int aio_read64( struct aiocb64 * const aiocbptr );
```

Arguments:

aiocbptr

A pointer to an asynchronous I/O control block of type [aiocb](#) or [aiocb64](#) (p. 141) that defines how much data to read, and from where.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *aio_read()* and *aio_read64()* functions asynchronously read *aiocbptr->aio_nbytes* from the file associated with *aiocbptr->aio_fildes* into the buffer pointed to by *aiocbptr->aio_buf*. The functions return when the read request has been initiated or queued to the file or device (even when the data can't be delivered immediately).

The asynchronous operation is submitted at the scheduling priority of the thread minus *aiocbp->aio_reqprio*.

You can pass the *aiocbptr* argument to [aio_error\(\)](#) (p. 127) and [aio_return\(\)](#) (p. 134), to determine the error status and return status of the asynchronous operation while it's proceeding. If an error condition is encountered during queuing, the functions return without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by *aio_offset*, as if [lseek\(\)](#) (p. 1555) were called immediately prior to the operation with an offset equal to *aio_offset* and a *whence* of `SEEK_SET`. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

These functions ignore the *aiocbptr->aio_lio_opcode* field.

If the buffer pointed to by *aiocbptr->aiobuf* or the control block pointed to by *aiocbptr* becomes an illegal address before the asynchronous I/O is completed, then the behavior is undefined.

Simultaneous asynchronous operations using the same *aiocbptr* produce undefined results.

If synchronized I/O is enabled on the file associated with *aiocbptr->aiofildes*, these functions behave in accordance with the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

If a system action changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the results of that action are undefined.

The following conditions may be detected synchronously at the time of the call to *aio_read()* or *aio_read64()*, or asynchronously. If any of these conditions are detected synchronously, these functions return -1 and set *errno* to the corresponding value. If any of these conditions are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value:

EBADF

The *aiocbptr->aiofildes* argument isn't a valid file descriptor that's open for reading.

EINVAL

The file offset value implied by *aiocbptr->aio_offset* would be invalid, *aiocbptr->aio_reqprio* isn't a valid value, or *aiocbptr->aio_nbytes* is invalid.

If *aio_read()* or *aio_read64()* successfully queues the I/O operation, but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by [read\(\)](#) (p. 2502). In addition, the error status of the asynchronous operation is set to one of the error statuses normally set *read()*, or one of the following:

EBADF

The *aiocbptr->aiofildes* isn't a valid file descriptor that's open for reading.

ECANCELED

The requested I/O was canceled before the I/O was completed, due to an explicit [aio_cancel\(\)](#) (p. 124), request.

EINVAL

The file offset value implied by *aiocbptr->aio_offset* would be invalid.

The following condition may be detected synchronously or asynchronously:

EOverflow

The file is a regular file, *aio_cbptr->aio_nbytes* is greater than 0, and the starting offset in *aio_cbptr->aio_offset* is before the end of the file and is at or beyond the offset maximum in the open file description associated with *aio_cbptr->aio_fildes*.

Returns:

0 if the I/O operation was successfully queued, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EAGAIN

The requested asynchronous I/O operation wasn't queued because of system resource limitations.

Classification:

aio_read() is [POSIX 1003.1 AIO](#) (p. 46); *aio_read64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an *aio_** function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aio_return()

Get the return status for an asynchronous I/O operation

Synopsis:

```
#include <aio.h>

ssize_t aio_return( struct aiocb * aiocbptr );
```

Arguments:

aiocbptr

A pointer to an asynchronous I/O control block of type [aiocb](#) (p. 141) whose return status you want to get.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *aio_return()* function returns the return status associated with the `aiocb` structure referenced by the *aiocbptr* argument. The return status for an asynchronous I/O operation is the value that's returned by the corresponding [read\(\)](#) (p. 2502), [write\(\)](#) (p. 3737), or [fsync\(\)](#) (p. 787) operation.

You can call *aio_return()* exactly once to retrieve the return status of a given asynchronous operation; if you use the same `aiocb` structure for the same operation in a call to *aio_return()* or [aio_error\(\)](#) (p. 127), an error may be returned.

Returns:

The value that's returned by the corresponding [read\(\)](#) (p. 2502), [write\(\)](#) (p. 3737), or [fsync\(\)](#) (p. 787) operation.

Errors:

EINVAL

The *aiocbptr* argument doesn't refer to an asynchronous operation whose return status hasn't yet been retrieved.

Classification:

POSIX 1003.1 AIO (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an *aio_** function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aio_suspend()

Wait for asynchronous I/O operations to be completed

Synopsis:

```
#include <aio.h>

int aio_suspend( const struct aiocb * const list[],
                 int nent,
                 const struct timespec * timeout );
```

Arguments:

list

A list of [aiocb](#) (p. 141) structures describing the asynchronous operations you want to wait for. Each `aiocb` structure must have been used in initiating an asynchronous I/O request via [aio_read\(\)](#) (p. 131), [aio_write\(\)](#) (p. 138), or [lio_listio\(\)](#) (p. 1519). The list may contain NULL pointers, which `aio_suspend()` ignores.

nent

The number of entries in the list.

timeout

NULL, or a pointer to a [timespec](#) (p. 3473) structure that specifies the maximum length of time you want to wait for.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `aio_suspend()` function suspends the calling thread until at least one of the asynchronous I/O operations referenced by the `list` argument has been completed, until a signal interrupts the function, or, if `timeout` isn't NULL, until the time interval specified by `timeout` has passed.

If any of the `aio_cb` structures in the list correspond to completed asynchronous I/O operations (i.e. the error status for the operation isn't `EINPROGRESS`) at the time of the call, `aio_suspend()` returns without suspending the calling thread.

Returns:

0 if one or more of the asynchronous I/O operations have been completed, otherwise -1 (`errno` (p. 539) is set).

You can determine which asynchronous I/O operations were completed by scanning the associated error and return status, using `aio_error()` (p. 127) and `aio_return()` (p. 134).

Errors:

EAGAIN

No asynchronous I/O operation indicated in the list was completed in the time interval indicated by `timeout`.

EINTR

A signal interrupted the `aio_suspend()` function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it's completed, this error return may be caused by the completion of one (or more) of the very I/O operations you're waiting on.

Classification:

[POSIX 1003.1 AIO](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an `aio_*` function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aio_write(), aio_write64()

Asynchronously write to a file

Synopsis:

```
#include <aio.h>

int aio_write( struct aiocb * aiocbptr );
int aio_write64( struct aiocb64 * aiocbptr );
```

Arguments:

aiocbptr

A pointer to an asynchronous I/O control block of type [aiocb](#) or [aiocb64](#) (p. 141) that defines how much data to write, and where to write it.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *aio_write()* and *aio_write64()* functions asynchronously write *aiocbptr->aio_nbytes* to the file associated with *aiocbptr->aio_fildes* from the buffer that *aiocbptr->aio_buf* points to. The functions return when the write request has been initiated or, at a minimum, queued to the file or device.

The asynchronous operation is submitted at the scheduling priority of the thread minus *aiocbptr->aio_reqprio*.

You can use the *aiocbptr* argument as an argument to [aio_error\(\)](#) (p. 127) and [aio_return\(\)](#) (p. 134) to determine the error status and return status of the asynchronous operation while it's in progress.

If `O_APPEND` isn't set for the file descriptor *aio_fildes*, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if [lseek\(\)](#) (p. 1555) were called immediately before the operation with an offset of *aio_offset* and a *whence* of `SEEK_SET`.

If `O_APPEND` is set for the file descriptor, write operations append to the file in the same order as the calls were made.

If synchronized I/O is enabled on the file associated with *aiocbptr->aio_fildes*, these functions behave in accordance with the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

The following conditions may be detected synchronously at the time of the call to *aio_write()* or *aio_write64()*, or asynchronously. If any of these conditions are detected synchronously, these functions return -1 and set *errno* to the corresponding value. If any of these conditions are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value:

EBADF

The *aiocbptr->aio_fildes* argument isn't a valid file descriptor that's open for writing.

EINVAL

The file offset value implied by *aiocbptr->aio_offset* would be invalid, *aiocbptr->aio_reqprio* isn't a valid value, or *aiocbptr->aio_nbytes* is invalid.

If *aio_write()* or *aio_write64()* successfully queues the I/O operation, but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by [write\(\)](#) (p. 3737). In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by *write()*, or one of the following:

EBADF

The *aiocbptr->aio_fildes* isn't a valid file descriptor that's open for writing.

ECANCELED

The requested I/O was canceled before the I/O was completed, due to an explicit [aio_cancel\(\)](#) (p. 124), request.

EINVAL

The file offset value implied by *aiocbptr->aio_offset* would be invalid.

The following condition may be detected synchronously or asynchronously:

EOVERFLOW

The file is a regular file, *aiocbptr->aio_nbytes* is greater than 0, and the starting offset in *aiocbptr->aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbptr->aio_fildes*.

Returns:

0 if the I/O operation was successfully queued, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EAGAIN**

The requested asynchronous I/O operation wasn't queued because of system resource limitations.

Classification:

aio_write() is [POSIX 1003.1 AIO](#) (p. 46); *aio_write64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The first time you call an *aio_** function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

aiocb, aiocb64

Asynchronous I/O control block

Synopsis:

```
#include <aio.h>

struct aiocb {
    int          aio_fildes;
    int          aio_reqprio;
    #if _FILE_OFFSET_BITS - 0 == 64
        off_t      aio_offset;
    #elif !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
    #if defined(__LITTLEENDIAN__)
        off_t      aio_offset;
        off_t      aio_offset_hi;
    #elif defined(__BIGENDIAN__)
        off_t      aio_offset_hi;
        off_t      aio_offset;
    #else
        #error endian not configured for system
    #endif
    #else
        #error _FILE_OFFSET_BITS value is unsupported
    #endif
    __volatile void* aio_buf;
    size_t          aio_nbytes;
    struct sigevent  aio_sigevent;
    int             aio_lio_opcode;
    ssize_t         _aio_reserved;
    int             _aio_pad[3];
    ...
};

#if _LARGEFILE64_SOURCE - 0 > 0
struct aiocb64 {
    int          aio_fildes;
    int          aio_reqprio;
    off64_t      aio_offset;
    __volatile void* aio_buf;
    size_t          aio_nbytes;
    struct sigevent  aio_sigevent;
    int             aio_lio_opcode;
    ssize_t         _aio_reserved;
    int             _aio_pad[3];
    ...
};
#endif /* _LARGEFILE64_SOURCE */
```

Description:

The `aiocb` and `aiocb64` structures define the control block for asynchronous I/O operations. They include at least the following:

aio_fildes

The file descriptor to use in an asynchronous I/O operation.



The *aio_** functions work with Transparent Distributed Processing; the file descriptor can be one that you've opened across Qnet.

aio_reqprio

The request priority offset.

aio_offset

The file offset.

aio_offset_hi

(aio_cb only) The high-order bits of the file offset.

aio_buf

A pointer to a buffer.

aio_nbytes

The length of a transfer.

aio_sigevent

A pointer to a [sigevent](#) (p. 2961) structure that specifies the signal number and value.

aio_lio_opcode

The operation to be performed; one of the following:

- LIO_NOP — a [lio_listio\(\)](#) (p. 1519) element operation option indicating that no transfer is requested.
- LIO_NOWAIT — a [lio_listio\(\)](#) synchronization operation indicating that the calling thread is to continue execution while the [lio_listio\(\)](#) operation is being performed, and no notification is given when the operation is complete.
- LIO_READ — a [lio_listio\(\)](#) element operation option requesting a read.
- LIO_WAIT — a [lio_listio\(\)](#) synchronization operation indicating that the calling thread is to suspend until the [lio_listio\(\)](#) operation is complete.
- LIO_WRITE — a [lio_listio\(\)](#) element operation option requesting a write.

Classification:

aio_cb is [POSIX 1003.1 AIO](#) (p. 46); aio_cb64 is [Large-file support](#) (p. 46)

Caveats:

The first time you call an *aiocb* function, a thread pool is created, making your process multithreaded if it isn't already. The thread pool isn't destroyed until your process ends.

alarm()

Schedule an alarm

Synopsis:

```
#include <unistd.h>

unsigned int alarm( unsigned int seconds );
```

Arguments:

seconds

The number of seconds of realtime to let elapse before raising the alarm, or zero to cancel any previous *alarm()* requests.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *alarm()* function causes the system to send the calling process a `SIGALRM` signal after a specified number of realtime seconds have elapsed. To add a handler for the signal, call [signal\(\)](#) (p. 2974) or [SignalAction\(\)](#) (p. 2977).



- Because of the nature of time measurement, the signal might actually get sent later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.
- Processor scheduling delays may cause the process to handle the signal after the desired time.

The *alarm()* requests aren't stacked; you can schedule only a single `SIGALRM` generation in this manner. If the `SIGALRM` hasn't yet been generated, *alarm()* reschedules the time at which the `SIGALRM` is generated.

Returns:

The number of seconds before the calling process is scheduled to receive a SIGALRM from the system, or zero if there was no previous *alarm()* request.

If an error occurs, an (unsigned) -1 is returned (*errno* (p. 539) is set).

Errors:**EAGAIN**

All timers are in use. You'll have to wait for a process to release one.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    unsigned int  timeleft;

    printf( "Set the alarm and sleep\n" );
    alarm( 10 );
    sleep( 5 ); /* go to sleep for 5 seconds */

    /*
     * To get the time left before the SIGALRM is
     * to arrive, one must cancel the initial timer,
     * which returns the amount of time it had
     * remaining.
     */
    timeleft = alarm( 0 );
    printf( "Time left before cancel, and rearm: %d\n",
           timeleft );

    /*
     * Start a new timer that kicks us when timeleft
     * seconds have passed.
     */

    alarm( timeleft );

    /*
     * Wait until we receive the SIGALRM signal; any
     * signal kills us, though, since we don't have
     * a signal handler.
     */
    printf( "Hanging around, waiting to die\n" );
    pause();
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

Requests from *alarm()*, *TimerAlarm()* (p. 3448), and *ualarm()* (p. 3550) aren't "stacked;" only a single SIGALRM generator can be scheduled with these functions. If the SIGALRM signal hasn't been generated, the next call to *alarm()*, *TimerAlarm()*, or *ualarm()* reschedules it.

alloca()

Allocate automatic space from the stack

Synopsis:

```
#include <alloca.h>

void* alloca( size_t size );
```

Arguments:

size

The number of bytes of memory to allocate.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `alloca()` function allocates space for an object of *size* bytes from the stack. The allocated space is automatically discarded when the current function exits.

-
- Don't use this function in an expression that's an argument to a function.
 - By default, `alloca()` is implemented as `__builtin_alloca()`. If you compile with the `-fno-builtin` option, you'll use the `libc` version of `alloca()`, which is a cover for [malloc\(\)](#) (p. 1569) and behaves in a slightly different manner:



- It keeps track of all blocks allocated with `alloca()` and reclaims any that are found to be deeper in the stack than the current invocation. It therefore doesn't reclaim storage as soon as it becomes invalid, but it will do so eventually.
- As a special case, `alloca(0)` reclaims storage without allocating any. You can use this feature to force garbage collection.

Returns:

A pointer to the start of the allocated memory, or `NULL` if an error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

FILE *open_err_file( char *name )
{
    char *buffer;

    /* allocate temporary buffer for file name */
    buffer = (char *)alloca( strlen( name ) + 5 );

    if( buffer ) {
        FILE *fp;

        sprintf( buffer, "%s.err", name );
        fp = fopen( buffer, "w" );

        return fp;
    }

    return (FILE *)NULL;
}

int main( void )
{
    FILE *fp;

    fp = open_err_file( "alloca_test" );
    if( fp == NULL ) {
        printf( "Unable to open error file\n" );
    } else {
        fprintf( fp, "Hello from the alloca test.\n" );
        fclose( fp );
    }

    return EXIT_SUCCESS;
}
```

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Don't use `alloca()` as an argument to a function.

alphasort()

Compare two directory entries

Synopsis:

```

#include <sys/types.h>
#include <sys/dir.h>

struct direct {
    unsigned long d_fileno;
    unsigned short d_reclen;
    unsigned short d_namlen;
    char d_name[1];
};

int alphasort( struct direct **d1,
               struct direct **d2);

```

Arguments:

d1, d2

Pointers to the `struct direct` directory entries that you want to compare.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `alphasort()` function alphabetically compares two directory entries. You can use it as the *compar* argument to [scandir\(\)](#) (p. 2711).

Returns:

< 0

The *d1* entry precedes the *d2* entry alphabetically.

0

The entries are equivalent.

> 0

The *d1* entry follows the *d2* entry alphabetically.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

_amblksiz

The increment for the break pointer

Synopsis:

```
#include <stdlib.h>

unsigned int _amblksiz
```

Description:

The `_amblksiz` global variable is the basic unit that's used for heap allocations to get memory from the system using `mmap()` (p. 1656). All underlying `mmap()` operations made by the heap allocator get memory as multiples of `_amblksiz`. By default, `_amblksiz` is set to $8 \times \text{PSIZ}$, or 32 KB. Its value must be a multiple of 4 KB, and currently is limited to being less than 256 KB.

In the current implementation of the allocator, requests for memory larger than 32 KB are automatically serviced by calling `mmap()` directly, while smaller allocations are serviced by a split-coalesce mechanism inside the allocator.

The value of `_amblksiz` affects all allocations that are smaller than 32 KB and require a core allocation. Memory that has become free will eventually return to the system when all memory associated with a specific core allocation has been released back to the allocator. Even when a block has been fully released to the allocator, it's possible for the allocator, for efficiency purposes, to retain some blocks locally within the heap (without releasing memory to the system immediately). This is done to avoid thrashing behavior, when requests to allocate and free memory cause the the allocator to constantly request and release memory to and from the system.

There are several ways that you can change `_amblksiz`:

- Redefine `_amblksiz` as a global in your code. Doing this overrides the definition from the library:

```
unsigned _amblksiz = 8*PSIZ;
```

No checking is done, but the value specified here must be a multiple of 4 KB. You can reset `_amblksiz` to a new value at any time, and this takes effect from that point onwards.



It isn't safe for a multithreaded program to dynamically change this value. Multithreaded applications should set this value as a global at compile time, and use the below mechanisms for safely manipulating `_amblksiz`.

- Set the **MALLOC_ARENA_SIZE** environment variable to the value you want to assign to `_ambksiz`. This is evaluated when `malloc()` is first initialised, so it's done before any allocations. The value specified is rounded up to the nearest multiple of 4 KB. Values above 0x40000 (256 KB) or below 4 KB aren't accepted at this time.
- Call `mallopt(MALLOC_ARENA_SIZE, size)`. This sets `_ambksiz` to the value specified by `size`. This is evaluated when you call [mallopt\(\)](#) (p. 1575). The value specified is rounded up to the nearest multiple of 4 KB. Values above 0x40000 (256 KB) or below 4 KB aren't accepted at this time.

If `size` is specified as zero, `mallopt()` returns the current value of `_ambksiz`.

Classification:

[QNX Neutrino](#) (p. 46)

_argc

The number of arguments passed to main()

Synopsis:

```
int _argc
```

Description:

This global variable holds the number of arguments passed to [main\(\)](#) (p. 1566).



This variable isn't defined in any header file. If you want to refer to it, you need to add your own extern statement.

Classification:

[QNX Neutrino](#) (p. 46)

_argv

A pointer to the vector of arguments passed to main()

Synopsis:

```
char ** _argv;
```

Description:

This global variable holds a pointer to a vector containing the actual arguments passed to [main\(\)](#) (p. 1566).



This variable isn't defined in any header file. If you want to refer to it, you need to add your own `extern` statement.

Classification:

[QNX Neutrino](#) (p. 46)

asctime(), asctime_r()

Convert time information to a string

Synopsis:

```
#include <time.h>

char* asctime( const struct tm* timeptr );
char* asctime_r( const struct tm* timeptr,
                 char* buf );
```

Arguments:

timeptr

A pointer to a `tm` structure that contains the time that you want to convert to a string.

buf

(*asctime_r()* only) A buffer in which *asctime_r()* can store the resulting string. This buffer must be large enough to hold at least 26 characters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *asctime()* and *asctime_r()* functions convert the time information in the structure pointed to by *timeptr* into a string containing exactly 26 characters, in the form:

```
Tue May  7 10:40:27 2002\n\0
```



The *asctime()* function places the result string in a static buffer that's reused every time you call *asctime()* or *ctime()* (p. 389). Calling *gmtime()* (p. 1005) or *localtime()* (p. 1531) could also change the date in this static buffer.

The result string for *asctime_r()* is contained in the buffer pointed to by *buf*.

All fields have a constant width. The newline character (`'\n'`) and a NUL character (`'\0'`) occupy the last two positions of the string.

Returns:

A pointer to the character string result, or NULL if an error occurred.

Classification:

asctime() is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *asctime_r()* is [POSIX 1003.1 TSF](#) (p. 46)

Table 1: *asctime()*

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Table 2: *asctime_r()*

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The *asctime()* and [ctime\(\)](#) (p. 389) functions place their results in a static buffer that's reused for each call to *asctime()* or *ctime()*.

asin(), asinf(), asinl()

Compute the arcsine of an angle

Synopsis:

```
#include <math.h>

double asin( double x );
float asinf( float x );
long double asinl( long double x );
```

Arguments:

x

The sine for which you want to find the angle.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the value of the arcsine (specified in radians) of *x*.

Returns:

The arcsine, in the range $(-\pi/2, \pi/2)$. For finite values not in the range $[-1,1]$, these functions return NaN. The return value for $\pm\text{Inf}$ is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", asin(.5) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
0.523599
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

asinh(), asinhf(), asinhl()

Compute the inverse hyperbolic sine

Synopsis:

```
#include <math.h>

double asinh( double x );
float asinhf( float x );
long double asinhl( long double x );
```

Arguments:

x

The value for which you want to compute the inverse hyperbolic sine.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the inverse hyperbolic sine of *x*.

Returns:

The inverse hyperbolic sine (specified in radians) of *x*.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", asinh( 0.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
0.481212
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

assert()

Print a diagnostic message and optionally terminate the program

Synopsis:

```
#include <assert.h>

void assert( int expression );
```

Arguments:

expression

Zero if you want to terminate the program; a nonzero value if you don't.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `assert()` macro prints a diagnostic message on the [stderr](#) (p. 3175) stream, and terminates the program, using [abort\(\)](#) (p. 54), if *expression* is false (0).

The diagnostic message includes the *expression*, the name of the source file (the value of `__FILE__`) and the line number of the failed assertion (the value of `__LINE__`).

No action is taken if *expression* is true (nonzero).

You typically use the `assert()` macro while developing a program, to identify program logic errors. You should choose the *expression* so that it's true when the program is functioning as intended.

After the program has been debugged, you can use the special “no debug” identifier, `NDEBUG`, to remove calls to `assert()` from the program when it's recompiled. If you use the `-D` option to `qcc` or a `#define` directive to define `NDEBUG` (with any value), the C preprocessor ignores all `assert()` calls in the program source.

To remove the calls to `assert()`, you must define `NDEBUG` in the code *before* including the `<assert.h>` header file (i.e. `#include <assert.h>`).

If you define `NDEBUG`, the preprocessor also ignores the expression you pass to `assert()`. For example, if your code includes:



```
assert((fd = open("filename", O_RDWR)) != -1);
```

and you define `NDEBUG`, the preprocessor ignores the entire call to `assert()`, including the call to `open()`.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void process_string( char *string )
{
    /* use assert to check argument */
    assert( string != NULL );
    assert( *string != '\0' );
    /* rest of code follows here */
}

int main( void )
{
    process_string( "hello" );
    process_string( "" );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

`assert()` is a macro.

atan(), atanf(), atanl()

Compute the arctangent of an angle

Synopsis:

```
#include <math.h>

double atan( double x );
float atanf( float x );
long double atanl( long double x );
```

Arguments:

x

The tangent for which you want to find the angle.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the arctangent (specified in radians) of *x*.

Returns:

The arctangent, in the range $(-\pi/2, \pi/2)$. For finite values of $|x| > 1$, these functions return NaN. The return value for $\pm\infty$ is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", atan(.5) );
    return EXIT_SUCCESS;
}
```

produces the output:

0.463648

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

atan2(), atan2f()

Compute the arctangent, determining the quadrant

Synopsis:

```
#include <math.h>

double atan2( double y,
              double x );

float atan2f( float y,
             float x );
```

Arguments:

x, y

The value (y/x) for which you want to find the angle.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the value of the arctangent (specified in radians) of y/x , using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero.

Returns:

The arctangent of y/x , in the range $(-\pi, \pi)$.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` (p. 539) to 0, call the function, and then check `errno` again. These functions don't change `errno` if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", atan2( .5, 1. ) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
0.463648
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

atanh(), atanhf(), atanhll()

Compute an inverse hyperbolic tangent

Synopsis:

```
#include <math.h>

double atanh( double x );
float atanhf( float x );
long double atanhll( long double x );
```

Arguments:

x

The value for which you want to compute the inverse hyperbolic tangent.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the inverse hyperbolic tangent (specified in radians) of *x*.

Returns:

The inverse hyperbolic tangent of *x*.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", atanh( 0.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
0.549306
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

atexit()

Register functions to be called during normal program termination

Synopsis:

```
#include <stdlib.h>

int atexit( register void (*func)(void) );
```

Arguments:

func

A pointer to the function you want to be called when the program terminates normally. This function has no arguments and doesn't return a value; its prototype should be:

```
void func( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atexit()` function registers a function to be called when the program terminates normally. If you register more than one function with `atexit()`, they're executed in a “last-in, first-out” order. Normal termination occurs either by a call to [exit\(\)](#) (p. 599) or a return from `main()`.

You can register a total of 32 functions with `atexit()`.



The functions registered with `atexit()` aren't called when the program terminates with a call to [_exit\(\)](#) (p. 596).

Returns:

0 for success, or nonzero if an error occurs.

Examples:

```
#include <stdio.h>
```

```
#include <stdlib.h>

void func1( void )
{
    printf( "last.\n" );
}

void func2( void )
{
    printf( "this " );
}

void func3( void )
{
    printf( "Do " );
}

int main( void )
{
    atexit( func1 );
    atexit( func2 );
    atexit( func3 );

    printf( "Do this first.\n" );

    return EXIT_SUCCESS;
}
```

produces the output:

```
Do this first.
Do this last.
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

atof()

Convert a string into a double

Synopsis:

```
#include <stdlib.h>

double atof( const char* ptr );
```

Arguments:

ptr

A pointer to the string to parse.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atof()` function converts the string pointed to by *ptr* to a `double`. Calling it is equivalent to calling [strtod\(\)](#) (p. 3253) like this:

```
strtod( ptr, (char**)NULL )
```

Returns:

The converted `double`, or `0.0` if an error occurs.

Errors:

If an error occurs, [errno](#) (p. 539) is set to `ERANGE`.

Examples:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    double x;

    x = atof( "3.1415926" );
    printf( "x = %F\n", x );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

atoi()

Convert a string containing a hexadecimal number into an unsigned number

Synopsis:

```
#include <stdlib.h>

unsigned atoi( const char* ptr );
```

Arguments:

ptr

A pointer to the string to parse.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *atoi()* function converts the string pointed to by *ptr* to unsigned representation, assuming the string contains a hexadecimal (base 16) number.

Returns:

The converted value.

Examples:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    unsigned x;

    x = atoi( "F1A6" );
    printf( "number is %x\n", x );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atoi()

Convert a string into an integer

Synopsis:

```
#include <stdlib.h>

int atoi( const char* ptr );
```

Arguments:

ptr

A pointer to the string to parse.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *atoi()* function converts the string pointed to by *ptr* to an `int`.

Returns:

The converted integer.

Examples:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int x;

    x = atoi( "-289" );
    printf( "x = %d\n", x );
    return EXIT_SUCCESS;
}
```

produces the output:

```
x = -289
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atol(), atoll()

Convert a string into a long integer

Synopsis:

```
#include <stdlib.h>

long atol( const char* ptr );
long long atoll( const char* ptr );
```

Arguments:

ptr

A pointer to the string to parse.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *atol()* function converts the string pointed to by *ptr* to a long integer; *atoll()* converts the string pointed to by *nptr* to a long long integer.

Returns:

atol()

A long integer.

atoll()

A long long integer.

Examples:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    long x;

    x = atol( "-289" );
    printf( "x = %d\n", x );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
x = -289
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_add()

Safely add to a variable

Synopsis:

```
#include <atomic.h>

void atomic_add( volatile unsigned * loc,
                unsigned incr );
```

Arguments:

loc

A pointer to the value that you want to add to.

incr

The number that you want to add.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_add()` function is a thread-safe way of doing an `(*loc) += incr` operation, even in a symmetric-multiprocessing system.

The `atomic_*`() functions are guaranteed to complete without being preempted by another thread. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*`() functions.

The `atomic_*`() functions are useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Examples:

To safely increment a counter shared between multiple threads:

```
#include <atomic.h>
...
volatile unsigned count;
...
atomic_add( &count, 1 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_add_value()

Safely add to a variable, returning the previous value

Synopsis:

```
#include <atomic.h>

unsigned atomic_add_value( volatile unsigned * loc,
                          unsigned incr );
```

Arguments:

loc

A pointer to the value that you want to add to.

incr

The number that you want to add.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_add_value()` function is a thread-safe way of doing an `(*loc) += incr` operation, even in a symmetric-multiprocessing system.



The `atomic_add_value()` function may be slower than [atomic_add\(\)](#) (p. 179).

The `atomic_*` functions are guaranteed to complete without being preempted by another thread. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*` functions.

The `atomic_*` functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Returns:

The previous value of *loc*'s contents.

Examples:

To safely increment a counter shared between multiple threads:

```
#include <atomic.h>
...
volatile unsigned count;
unsigned previous;
...
previous = atomic_add_value( &count, 1 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_clr()

Safely clear a variable

Synopsis:

```
#include <atomic.h>

void atomic_clr( volatile unsigned * loc,
                unsigned bits );
```

Arguments:

loc

A pointer to the value that you want to clear bits in.

bits

The bits that you want to clear.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *atomic_clr()* function is a thread-safe way of doing an `(*loc) &= ~bits` operation.

The *atomic_**() functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the *atomic_**() functions.

The *atomic_**() functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Examples:

To safely clear the 0x10101010 bits in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
...
atomic_clr( &flags, 0x10101010 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_clr_value()

Safely clear a variable, returning the previous value

Synopsis:

```
#include <atomic.h>

unsigned atomic_clr_value( volatile unsigned * loc,
                          unsigned bits );
```

Arguments:

loc

A pointer to the value that you want to clear bits in.

bits

The bits that you want to clear.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_clr_value()` function is a thread-safe way of doing an `(*loc) &= ~bits` operation.



The `atomic_clr_value()` function may be slower than [atomic_clr\(\)](#) (p. 183).

The `atomic_*` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*` functions.

The `atomic_*` functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Returns:

The previous value of *loc*'s contents.

Examples:

To safely clear the 0x10101010 bits in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...
previous = atomic_clr_value( &flags, 0x10101010 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_set()

Safely set bits in a variable

Synopsis:

```
#include <atomic.h>

void atomic_set( volatile unsigned * loc,
                unsigned bits );
```

Arguments:

loc

A pointer to the location whose bits you want to set.

bits

The bits that you want to set.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_set()` function is a thread-safe way of doing an `(*loc) |= bits` operation.

The `atomic_*`() functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*`() functions.

The `atomic_*`() functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Examples:

To safely set the 1 bit in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
...
atomic_set( &flags, 0x01 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_set_value()

Safely set bits in a variable, returning the previous value

Synopsis:

```
#include <atomic.h>

unsigned atomic_set_value( volatile unsigned * loc,
                          unsigned bits );
```

Arguments:

loc

A pointer to the location whose bits you want to set.

bits

The bits that you want to set.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_set_value()` function is a thread-safe way of doing an `(*loc) |= bits` operation.



The `atomic_set_value()` function may be slower than `atomic_set()` (p. 187).

The `atomic_*` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*` functions.

The `atomic_*` functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Returns:

The previous value of *loc*'s contents.

Examples:

To safely set the 1 bit in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...
previous = atomic_set_value( &flags, 0x01 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_sub()

Safely subtract from a variable

Synopsis:

```
#include <atomic.h>

void atomic_sub( volatile unsigned * loc,
                unsigned decr );
```

Arguments:

loc

A pointer to the value that you want to subtract from.

decr

The number that you want to subtract.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_sub()` function is a thread-safe way of doing a `(*loc) -= decr` operation, even in a symmetric-multiprocessing system.

The `atomic_*` functions are guaranteed to complete without being preempted by another thread. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*` functions.

The `atomic_*` functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Examples:

Safely subtract 1 from a counter:

```
#include <atomic.h>
...
volatile unsigned count;
...
atomic_sub( &count, 1 );
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_sub_value()

Safely subtract from a variable, returning the previous value

Synopsis:

```
#include <atomic.h>

unsigned atomic_sub_value( volatile unsigned * loc,
                          unsigned  decr );
```

Arguments:

loc

A pointer to the value that you want to subtract from.

decr

The number that you want to subtract.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_sub_value()` function is a thread-safe way of doing a `(*loc) -= decr` operation, even in a symmetric-multiprocessing system.



The `atomic_sub_value()` function may be slower than [atomic_sub\(\)](#) (p. 191).

The `atomic_*` functions are guaranteed to complete without being preempted by another thread. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*` functions.

The `atomic_*` functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Returns:

The previous value of *loc*'s contents.

Examples:

Safely subtract 1 from a counter:

```
#include <atomic.h>
...
volatile unsigned count;
unsigned previous;
...
previous = atomic_sub_value( &count, 1 );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_toggle()

Safely toggle a variable

Synopsis:

```
#include <atomic.h>

void atomic_toggle( volatile unsigned * loc,
                   unsigned bits );
```

Arguments:

loc

A pointer to the location whose bits you want to toggle.

bits

The bits that you want to change.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_toggle()` function is a thread-safe way of doing an `(*loc) ^= bits` operation.

The `atomic_*`() functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*`() functions.

The `atomic_*`() functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Examples:

To safely toggle the 0xdeadbeef bits in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
...
atomic_toggle( &flags, 0xdeadbeef );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

atomic_toggle_value()

Safely toggle a variable, returning the previous value

Synopsis:

```
#include <atomic.h>

unsigned atomic_toggle_value(
    volatile unsigned * loc,
    unsigned bits );
```

Arguments:

loc

A pointer to the location whose bits you want to toggle.

bits

The bits that you want to change.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_toggle_value()` function is a thread-safe way of doing an `(*loc) ^= bits` operation.



The `atomic_toggle_value()` function may be slower than [atomic_toggle\(\)](#) (p. 195).

The `atomic_*` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system. When modifying a variable shared between a thread and an interrupt, you *must* either disable interrupts or use the `atomic_*` functions.

The `atomic_*` functions are also useful for modifying variables that are referenced by more than one thread (that aren't necessarily in the same process) without having to use a mutex.



Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

Returns:

The previous value of *loc*'s contents.

Examples:

To safely toggle the 0xdeadbeef bits in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...
previous = atomic_toggle_value( &flags, 0xdeadbeef );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

_auxv

A pointer to a vector of auxiliary arguments to main()

Synopsis:

```
auxv_t * _auxv;
```

Description:

This global variable holds a pointer to a vector of auxiliary arguments to [main\(\)](#) (p. 1566). For more information, see `<sys/auxv.h>`.



This variable isn't defined in any header file. If you want to refer to it, you need to add your own `extern` statement.

Classification:

[QNX Neutrino](#) (p. 46)

Chapter 4

B

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following have been deprecated:

Instead of:	Use:
<i>barrier_attr_destroy()</i>	pthread_barrierattr_destroy() (p. 2243)
<i>barrier_attr_getpshared()</i>	pthread_barrierattr_getpshared() (p. 2245)
<i>barrier_attr_init()</i>	pthread_barrierattr_init() (p. 2247)
<i>barrier_attr_setpshared()</i>	pthread_barrierattr_setpshared() (p. 2249)
<i>barrier_destroy()</i>	pthread_barrier_destroy() (p. 2237)
<i>barrier_init()</i>	pthread_barrier_init() (p. 2239)
<i>barrier_wait()</i>	pthread_barrier_wait() (p. 2241)

This chapter describes the API components whose names begin with “B”.

basename()

Find the part of a string after the last slash (/)

Synopsis:

```
#include <libgen.h>

char* basename( char* path );
```

Arguments:

path

The string to parse.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *basename()* function takes the pathname pointed to by *path* and returns a pointer to the final component of the pathname, deleting any trailing “/” characters.

The *basename()* function returns:

A pointer to the string “/”

If the string consists entirely of the “/” character

A pointer to the string “.”

If *path* is a NULL pointer, or points to an empty string

Together the [dirname\(\)](#) (p. 424) and *basename()* functions yield a complete pathname. The expression *dirname(path)* obtains the pathname of the directory where *basename(path)* is found.



The *basename()* function might modify the string pointed to by *path*, and can return a pointer to static storage.

Returns:

A pointer to the final component of *path*.

Examples:

```

#include <stdio.h>
#include <libgen.h>
#include <stdlib.h>

int main( int argc, char** argv )
{
    int x;
    for( x = 1; x < argc; x++ ) {
        printf( "%s\n", basename( argv[x] ) );
    }
    return EXIT_SUCCESS;
}

```

The table below shows the output of the program, given the input:

Input	Output
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

bcmp()

Compare a given number of characters in two strings

Synopsis:

```
#include <strings.h>

int bcmp( const void *s1,
          const void *s2,
          size_t n );
```

Arguments:

s1, s2

The strings you want to compare.

n

The number of bytes to compare.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *bcmp()* function compares the byte string pointed to by *s1* to the string pointed to by *s2*. The number of bytes to compare is specified by *n*. NUL characters may be included in the comparison.



This function is similar to the ANSI [*memcmp\(\)*](#) (p. 1609) function, but tests only for equality. New code should use the ANSI function.

Returns:

0

The byte strings are identical.

1

The byte strings aren't identical.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
    if( bcmp( "Hello there", "Hello world", 6 ) ) {
        printf( "Not equal\n" );
    } else {
        printf( "Equal\n" );
    }
    return EXIT_SUCCESS;
}
```

produces the output:

Equal

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

bcopy()

Copy a number of characters in one string to another

Synopsis:

```
#include <strings.h>

void bcopy( const void *src,
            void *dst,
            size_t n );
```

Arguments:

src

The string you want to copy.

dst

An existing array into which you want to copy the string.

n

The number of bytes to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `bcopy()` function copies the byte string pointed to by `src` (including any NUL characters) into the array pointed to by `dst`. The number of bytes to copy is specified by `n`. Copying of overlapping objects is guaranteed to work properly.



This function is similar to the ANSI [memmove\(\)](#) (p. 1618) function, but the order of arguments is different. New code should use the ANSI function.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
```

```
    auto char buffer[80];

    bcopy( "Hello ", buffer, 6 );
    bcopy( "world", &buffer[6], 6 );
    printf( "%s\n", buffer );
    return EXIT_SUCCESS;
}
```

produces the output:

```
Hello world
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

bind()

Bind a name to a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind( int s,
          const struct sockaddr * name,
          socklen_t namelen );
```

Arguments:

s

The file descriptor to be bound.

name

A pointer to the `sockaddr` structure that holds the address to be bound to the socket. The socket length and format depend upon its address family.

namelen

The length of the `sockaddr` structure pointed to by *name*.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

When a socket is created with [socket\(\)](#) (p. 3075), it exists in a namespace (address family) but has no name assigned to it. The *bind()* function assigns a name to that unnamed socket.



The *bind()* function assigns a local address. Use [connect\(\)](#) (p. 348) to assign a remote address.

The rules used for binding names vary between communication domains.



Before calling *bind()* on an AF_INET socket, set the *af_family* member of the *sockaddr* structure to AF_INET. Up until QNX Neutrino 6.4.0, a value of 0 was accepted and assumed to be this value.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EACCES

The requested address is protected, and the current user has inadequate permission to access it.

EADDRINUSE

The specified address is already in use.

EADDRNOTAVAIL

The specified address isn't available from the local machine.

EBADF

Invalid descriptor *s*.

EFAULT

The *name* parameter isn't in a valid part of the user address space.

EINVAL

The socket is already bound to an address.

ENOTSOCK

The given file descriptor isn't for a socket.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

bindresvport()

Bind a socket to a privileged IP port

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>

int bindresvport( int sd,
                  struct sockaddr_in * sin );
```

Arguments:

sd

The socket descriptor to bind to the port.

sin

A pointer to a `sockaddr_in` structure that specifies the privileged IP port.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *bindresvport()* function binds a socket descriptor to a privileged IP port (i.e. a port number in the range 0-1023).



Only `root` can bind to a privileged port; this call fails for any other user.

Returns:

0

Success.

-1

An error occurred ([*errno*](#) (p. 539) is set).

Errors:

EACCES

You must be root to call *bindresvport()*.

EADDRINUSE

The specified address is already in use.

EADDRNOTAVAIL

The specified address isn't available from the local machine.

EBADF

Invalid descriptor *sd*.

EFAULT

The *sin* parameter isn't a valid pointer to a `sockaddr_in` structure.

EINVAL

The socket is already bound to a port.

EPFNOSUPPORT

The protocol family isn't supported.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

brk()

Change the amount of space allocated for the calling process's data segment

Synopsis:

```
#include <unistd.h>

int brk( void* endds );
```

Arguments:

endds

A pointer to the new end of the data segment.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `brk()` function is used to change dynamically the amount of space allocated for the calling process's data segment (see the `exec*` functions).

The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process, its contents are undefined.

When a program begins execution using `execve()` (p. 579), the break is set at the highest location defined by the program and data storage areas.

You can call `getrlimit()` (p. 942) to determine the maximum permissible size of the data segment; it isn't possible to set the break beyond the `rlim_max` value returned from `getrlimit()`:

```
end + rlim.rlim_max
```

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).**Errors:****ENOMEM**

This could mean:

- The data segment size limit, as set by [setrlimit\(\)](#) (p. 2899), would be exceeded.
- The maximum possible size of a data segment (compiled into the system) would be exceeded.
- Out of address space; the new break value would extend into an area of the address space defined by some previously established mapping (see [mmap\(\)](#) (p. 1656)).

EAGAIN

The total amount of system memory available for private pages is temporarily insufficient. This may occur even though the space requested was less than the maximum data segment size.

Classification:[Legacy Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Don't use [brk\(\)](#) and [sbrk\(\)](#) with any other memory functions (such as [malloc\(\)](#) (p. 1569), [mmap\(\)](#) (p. 1656), and [free\(\)](#) (p. 717)). The [brk\(\)](#) function assumes that the heap is contiguous; in QNX Neutrino, memory is returned to the system by the heap, causing the heap to become sparse. The QNX Neutrino [malloc\(\)](#) function is based on [mmap\(\)](#), and not on [brk\(\)](#).

The *brk()* function has been used in specialized cases where no other memory allocation function provided the same capability. Use *mmap()* instead because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

The value of the argument to *brk()* is rounded up for alignment with eight-byte boundaries.

bsearch()

Perform a binary search on a sorted array

Synopsis:

```
#include <stdlib.h>

void *bsearch( const void *key,
               const void *base,
               size_t num,
               size_t width,
               int (*compar)( const void *pkey,
                              const void *pbase) );
```

Arguments:

key

The object to search for.

base

A pointer to the first element in the array.

num

The number of elements in the array.

width

The size of an element, in bytes.

compare

A pointer to a user-supplied function that *bsearch()* calls to compare an array element with the *key*.

The arguments to the comparison function are:

- *pkey* — the same pointer as *key*
- *pbase* — a pointer to an element in the array.

The comparison function must return an integer less than, equal to, or greater than zero if the *key* object is less than, equal to, or greater than the element in the array.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `bsearch()` function performs a binary search on the sorted array of `num` elements pointed to by `base`, for an item that matches the object pointed to by `key`.

Returns:

A pointer to a matching member of the array, or `NULL` if a matching object couldn't be found.



If there are multiple values in the array that match the `key`, the return value could be any of these duplicate values.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const char *keywords[] = {
    "auto",
    "break",
    "case",
    "char",
    /* ... */
    "while"
};

#define NUM_KW    sizeof(keywords) / sizeof(char *)

int kw_compare( const void *p1, const void *p2 )
{
    const char *p1c = (const char *) p1;
    const char **p2c = (const char **) p2;

    return( strcmp( p1c, *p2c ) );
}

int keyword_lookup( const char *name )
{
    const char **key;

    key = (char const **) bsearch( name, keywords,
                                   NUM_KW, sizeof( char * ), kw_compare );
    if( key == NULL ) return( -1 );

    return key - keywords;
}

int main( void )
{
    printf( "%d\n", keyword_lookup( "case" ) );
    printf( "%d\n", keyword_lookup( "crigger" ) );
    printf( "%d\n", keyword_lookup( "auto" ) );

    return EXIT_SUCCESS;
}
```

This program produces the following output:

```
2
-1
0
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

bt_get_backtrace()

Collect a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_get_backtrace( bt_accessor_t *acc,
                     bt_addr_t *addrs,
                     int len );
```

Arguments:

acc

A pointer to a `bt_accessor_t` structure. This is an opaque structure that holds the identity of the thread to backtrace.

addrs

An array of `bt_addr_t` members where the function can store the addresses; `addrs[0]` is the top of stack.

len

The maximum number of addresses to collect and store in *addrs*.

Library:

libbacktrace

Use the `-l backtrace` option to `qcc` to link against this library.

Description:

The `bt_get_backtrace()` function collects a backtrace up to *len* deep. The behavior depends on how you initialized the accessor when you called `bt_init_accessor()` (p. 221):

- If `BT_SELF`, the library uses direct memory access to gather the information from the stack.
- If `BT_THREAD`, or for `BT_PROCESS` where the process ID and thread ID are for the calling thread, the library uses direct memory access.

- For `BT_THREAD` where the thread ID isn't for the calling thread, or for `BT_PROCESS` with the process ID of the calling process and the thread ID other than that of the calling thread, the library stops the thread, fetches the thread context and registers, uses direct memory access, and then restarts the thread.
- For `BT_PROCESS` where the process ID isn't that of the calling process, the library stops the thread, fetches the thread context and registers, uses `/proc/pid/` as to access memory, and then restarts the thread.

Note the following:

- When you're backtracing within the same process, if `bt_get_backtrace()` attempts to access a non-existent memory location, signals are masked, and `SIGSEGV` is trapped to avoid crashing the program.
- QNX Neutrino 6.3.0 and earlier can't stop a single thread, so in this case stopping a thread really means stopping all threads in the process.
- QNX Neutrino 6.3.2 and later can stop a single thread.
- If other applications and processes start and stop threads, it may interfere with `bt_get_backtrace()`.

Returns:

The number of addresses stored in *addrs*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

bt_init_accessor()

Initialize a backtrace accessor



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_init_accessor( bt_accessor_t *acc,
                    bt_acc_type_t type, ... );
```

Arguments:

acc

A pointer to a `bt_accessor_t` structure. This is an opaque structure that holds the identity of the thread to backtrace.

type

The type of backtrace to perform; one of:

- `BT_SELF` — backtrace the calling thread.
- `BT_THREAD` — backtrace a specified thread in the current process.
- `BT_PROCESS` — backtrace a specified thread in another process.

Additional arguments are required for `BT_THREAD` and `BT_PROCESS`, as described below.

Library:

`libbacktrace`

Use the `-l backtrace` option to `qcc` to link against this library.

Description:

The `bt_init_accessor()` function initializes the accessor with the identity of the thread to backtrace:

- To backtrace the calling thread, use:

```
bt_init_accessor(bt_accessor_t *acc, BT_SELF);
```

- To backtrace a thread in the current process, use:

```
bt_init_accessor(bt_accessor_t *acc, BT_THREAD, pthread_t tid);
```

- To backtrace a thread in any process, use:

```
bt_init_accessor(bt_accessor_t *acc, BT_PROCESS, pid_t pid, pthread_t tid)
```



The *bt_acc_self* global variable is a preinitialized accessor used to backtrace the current thread. Don't call *bt_init_accessor()* or *bt_release_accessor()* for this variable.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *acc* argument is NULL, or the *type* is invalid.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

bt_load_memmap()

Load a memory map associated with a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_load_memmap( bt_accessor_t *acc,
                   bt_memmap_t * memmap );
```

Arguments:

acc

A pointer to a `bt_accessor_t` structure. This is an opaque structure that holds the identity of the thread to backtrace.

memmap

A pointer to a location where the function can store the memory map.

Library:

`libbacktrace`

Use the `-l backtrace` option to `qcc` to link against this library.

Description:

The `bt_load_memmap()` function reads the memory map information from the process represented by *acc*, and saves the information in *memmap*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acc* or *memmap* argument is NULL.

ENOMEM

There wasn't enough memory to load the memory map.

This function can also set *errno* to any of the values that [devctl\(\)](#) (p. 403) and [open\(\)](#) (p. 1905) can.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

bt_release_accessor()

Release an accessor for a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_release_accessor(bt_accessor_t *acc);
```

Arguments:

acc

A pointer to a `bt_accessor_t` structure. This is an opaque structure that holds the identity of the thread to backtrace.

Library:

`libbacktrace`

Use the `-l backtrace` option to `qcc` to link against this library.

Description:

The `bt_release_accessor()` function releases all resources that `acc` keeps track of.



Don't release an accessor while another function is using it.

Don't call `bt_init_accessor()` or `bt_release_accessor()` for the `bt_acc_self` global variable.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *acc* argument is **NULL**.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

bt_set_flags()

Set or clear the flags for backtracing



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_set_flags( bt_accessor_t *acc,
                 unsigned flags,
                 int onoff );
```

Arguments:

acc

A pointer to a `bt_accessor_t` structure. This is an opaque structure that holds the identity of the thread to backtrace.

flags

A bitwise ORing of the flags you want to set:

- `BTF_LIVE_BACKTRACE` — typically [*bt_get_backtrace\(\)*](#) (p. 219) freezes a thread before gathering the backtrace. If you set this flag, *bt_get_backtrace()* doesn't freeze the thread. This can be useful if you want to freeze many threads at the same time, and then collect the backtrace for all threads, thus providing a coherent snapshot for all of the threads.

onoff

Zero to clear the specified flags, or a nonzero value to set the flags.

Library:

`libbacktrace`

Use the `-l backtrace` option to `qcc` to link against this library.

Description:

The *bt_set_flags()* function lets you turn the specified flags on or off.

Returns:**0**

Success.

-1An error occurred ([errno](#) (p. 539) is set).**Errors:****EINVAL**The *flags* argument specifies an invalid flag.**Classification:**[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

bt_sprn_memmap()

Format the memory map information for a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_sprn_memmap( bt_memmap_t *memmap,
                   char *out,
                   size_t outlen );
```

Arguments:

memmap

A pointer to memory-map information for the process you've collected backtracing for. Use [bt_load_memmap\(\)](#) (p. 223) to initialize this variable.

out

A string where the function can store the formatted memory map.

outlen

The size of *out*.

Library:

libbacktrace

Use the -l backtrace option to qcc to link against this library.

Description:

The *bt_sprn_memmap()* function formats the memory map information into *out*.

TBD: The exact format may change.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

bt_sprnf_addrs()

Format the addresses from a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

int bt_sprnf_addrs( bt_memmap_t *memmap,
                   bt_addr_t *addrs,
                   int addrslen,
                   char *fmt,
                   char *out,
                   size_t outlen,
                   char *separator );
```

Arguments:

memmap

NULL, or a pointer to memory-map information for the process you've collected backtracing for. Use [bt_load_memmap\(\)](#) (p. 223) to initialize this variable.

addrs

An array of addresses that you want to format.

addrslen

The number of entries in the *addrs* array.

fmt

The format to use for each entry; see below.

out

A pointer to a buffer where the function can store the formatted output.

outlen

The length of the *out*, in bytes.

separator

NULL, or a pointer to a separator to write between the formatted addresses.

Library:

libbacktrace

Use the `-l backtrace` option to `qcc` to link against this library.

Description:

The `bt_sprnf_addr()` function formats the addresses in the `addrs` using `fmt` as the format for each entry, and storing the formatted addresses in the `out` buffer. If `separator` isn't NULL, the function writes it in `out` between each formatted address.

The `memmap` is optional, but if specified, it allows access to more formatting styles. The format for each entry (`fmt`), is analogous to [printf\(\)](#) (p. 2133) formats:

- `%%` — a literal %.
- `%a` — an address in memory.
- `%l` — an address relative to the start of the file it belongs to (i.e. this corresponds to the address in the object file).
- `%o` — an address offset from the beginning of the object files. (i.e. `%a == %l + %o`).
- `%f` — an object file that the address belongs to (e.g. `/some/dir/libc.so`).
- `%I` — a numerical index that corresponds to the numerical index in the memory map printouts (i.e. if the memory map is printed, then using `%I` instead of `%f` in the backtrace minimizes the size of the backtrace).



For the `%l`, `%o`, `%f`, and `%I` formats, `memmap` must not be NULL.

Returns:

The number of addresses from `addr` that could be completely formatted and written in `out`, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EINVAL**

One or more of the arguments is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

bt_translate_addr()

Translate the addresses from a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

void bt_translate_addr( bt_memmap_t *memmap,
                      bt_addr_t *addr,
                      int addrlen,
                      bt_addr_t *reladdr,
                      bt_addr_t *offsets,
                      int *index,
                      char **filenames );
```

Arguments:

memmap

A pointer to memory-map information for the process you've collected backtracing for. Use [bt_load_memmap\(\)](#) (p. 223) to initialize this variable.

addr

An array of addresses that you want to translate.

addrlen

The number of entries in the *addr* array, as well as the number of entries in the *reladdr*, *offsets*, *index*, and *filenames* arrays (if non-NULL).

reladdr

NULL, or an array where the function can store the addresses as specified in the object files.

offsets

NULL, or an array where the function can store the difference between the in-file address and in-memory address (i.e. *offsets*[0] = *addr*[0] - *mod_addr*[0]).

index

NULL, or an array where the function can store the memory map index for each address.

filenames

NULL, or an array where the function can store the object file name for each address.



The file names aren't copied from the memory map, so they're valid only until you unload the memory map.

Library:

libbacktrace

Use the -l backtrace option to qcc to link against this library.

Description:

The *bt_translate_addrs()* function translates all process addresses in *addrs*. You can pass a NULL pointer for any of the result arrays that you don't need.



Programs typically use [bt_sprnf_addrs\(\)](#) (p. 231) instead of *bt_translate_addrs()*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

bt_unload_memmap()

Unload a memory map associated with a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

Synopsis:

```
#include <backtrace.h>

void bt_unload_memmap( bt_memmap_t *memmap );
```

Arguments:

memmap

A pointer to the memory map that you want to unload.

Library:

libbacktrace
Use the -l backtrace option to qcc to link against this library.

Description:

The *bt_unload_memmap()* function releases the memory that *memmap* holds. Don't use *memmap* after invoking this function.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_btext

The beginning of the text segment

Synopsis:

N/A

Description:

This linker symbol defines the beginning of the text segment. This variable isn't defined in any header file.

Classification:

[QNX Neutrino](#) (p. 46)

btowc()

Convert a single-byte character to a wide character

Synopsis:

```
#include <wchar.h>

wint_t btowc( int c );
```

Arguments:

c

The single-byte character that you want to convert.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *btowc()* function converts the given character (if it's a valid one-byte character in the initial shift state) into a wide character.

This function is the single-byte version of [mbtowc\(\)](#) (p. 1595).

Returns:

The wide-character representation of the character, or WEOF if *c* has the value EOF or (unsigned char) *c* isn't a valid one-byte character in the initial conversion state.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

bzero()

Set the first part of an object to null bytes

Synopsis:

```
#include <strings.h>

void bzero( void *dst,
            size_t n );
```

Arguments:

dst

An existing object that you want to fill with zeroes.

n

The number of bytes to fill.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *bzero()* function fills the first *n* bytes of the object pointed to by *dst* with zero (NUL) bytes.



This function is similar to the ANSI [memset\(\)](#) (p. 1620) function. New code should use the ANSI function.

Examples:

```
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char buffer[80];

    bzero( buffer, 80 );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Chapter 5

C

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<code>curl_*</code>	http://curl.haxx.se/libcurl/

This chapter describes the API components whose names begin with “C”.

cabs()*, *cabsf()

Compute the absolute value of a complex number

Synopsis:

```
#include <math.h>

struct __cabsargs {
    double x; /* real part */
    double y; /* imaginary part */
};

double cabs( struct __cabsargs value );

struct __cabsfargs {
    float x; /* real part */
    float y; /* imaginary part */
};

float cabsf( struct __cabsfargs value );
```

Arguments:

value

The complex value that you want to get the absolute value of.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the absolute value of the complex number specified by *value*, using a calculation equivalent to:

```
sqrt( ( value.x * value.x ) + ( value.y * value.y ) );
```

Returns:

The absolute value of *value*.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

struct __cabsargs c = { -3.0, 4.0 };
```

```
int main( void )
{
    printf( "%f\n", cabs( c ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
5.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

cache_fini()

Free cache-coherency resources when the driver is unloaded

Synopsis:

```
int cache_fini(struct cache_ctrl *cinfo);
```

Arguments:

cinfo

A pointer to the structure that was originally passed to *cache_init()*. See the [cache_init\(\)](#) (p. 248) function.

Library:

libcache

Use the -l cache option to qcc to link against this library.

Description:

Call the *cache_fini()* function to free up any resources that were allocated by the *cache_init()* function. The *cinfo* is a pointer to the structure that was originally passed to *cache_init()*.



This function was added in QNX Momentics 6.3.0 SP2.

Returns:

0

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

CACHE_FLUSH()

Flush cache lines associated with a data buffer

Synopsis:

```
#include <sys/cache.h>

CACHE_FLUSH(struct cache_ctrl *cinfo,
             void *vaddr,
             uint64_t paddr,
             size_t len);
```

Arguments:

cinfo

A pointer to the structure that was initially passed to *cache_init()*.

vaddr

The virtual address of the buffer; this is a pointer to the data in the driver's virtual address space.

paddr

The physical address of the buffer: this is typically in the same address space that the external device will use to reference the data. The physical address is obtained by calling *mem_offset64()*. Since this function is fairly costly, drivers typically allocate a pool of data buffers at initialization (e.g. by calling *mmap()* with the MAP_PHYS and MAP_ANON flags), and predetermine the physical addresses of the data.

len

The number of bytes in the buffer, for which cached data should be flushed to memory.

Library:

libcache

Use the -l cache option to qcc to link against this library.

Description:

This macro is used to flush any cache lines associated with a data buffer out to memory. This routine ensures that any modifications that have been made to the data by the

CPU will be reflected by the contents of memory, and thus an external device reading the data won't retrieve stale data. For more information about cache coherency, see the entry for [cache_init\(\)](#) (p. 248).



This macro was added in QNX Momentics 6.3.0 SP2.

Before using the `CACHE_*` macros on ARM platforms, the calling thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- obtain I/O privileges by calling [ThreadCtl\(\)](#) (p. 3408), specifying the `_NTO_TCTL_IO` flag:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

Failing to do so may result in a `SIGILL` (illegal instruction) signal.

Environment variables:

The following environment variables, if they exist, affect the behavior of this function:

CACHE_NOP

Instructs the library that the [CACHE_FLUSH\(\)](#) (p. 246) and [CACHE_INVALID\(\)](#) (p. 253) macros should have no effect.

CACHE_MSINC

Instructs the library that the [CACHE_FLUSH\(\)](#) (p. 246) and [CACHE_INVALID\(\)](#) (p. 253) macros should use the [msync\(\)](#) (p. 1825) C library call to perform cache synchronization.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

`CACHE_FLUSH()` is implemented as a macro.

cache_init()

Register with the cache-coherency library

Synopsis:

```
#include <sys/cache.h>

int cache_init(int flags,
               struct cache_ctrl *cinfo,
               const char *dllname);
```

Arguments:

flags

Zero, or flags that control the behavior of the cache-coherency library. The only flag currently defined is:

- **CACHE_INIT_FLAG_IGNORE_SCAN** — Specify that memory accesses to and from the device aren't snooped, whether or not the data caches in the system are reported as snooping or not. This might be required for devices in the system that bypass the bus-snooping mechanism, or as a workaround for hardware bugs.

cinfo

A pointer to a `cache_ctrl` structure that contains state information. The library uses this structure to store various information that uses when performing synchronization operations. The driver should allocate and initialize this structure. All of the members of the structure should be initialized with zeros, with the exception of the *fd* field. For more information regarding the members of this structure, see the description section.

dllname

The path of a DLL to load that provides cache-synchronization routines or NULL to use the default routines.

Library:

`libcache`

Use the `-l cache` option to `qcc` to link against this library.

Description:

The `cache_init()` function initializes the cache coherency library (Libcache). Your driver must call `cache_init()` before using the library.



This function was added in QNX Momentics 6.3.0 SP2.

Members of the `cache_ctrl` structure

The `cache_ctrl` structure includes at least the following members:

cache_line_size

When this function returns, this field will specify the size, in bytes, of a cache line's worth of data. If the system implements a bus-snooping protocol, this field may contain zero.

cache_flush_rate

Provides a runtime indication to the driver, of the cost of flushing the cache:

CACHE_OP_RATE_SNOOP

Due to a bus-snooping mechanism, a cache flush operation has negligible cost.

CACHE_OP_RATE_INLINE

Cache flush operations are implemented with CPU-specific inline routines, and are inexpensive.

CACHE_OP_RATE_CALLOUT

Cache flush operations are implemented by calling an external function, which incurs a small CPU overhead.

CACHE_OP_RATE_MSINC

Cache flush operations are implemented by calling [*msync\(\)*](#) (p. 1825). Since this function is implemented with a system call, the operation will be very expensive. It is very unlikely that the library will end up calling *msync()*, but in the event that it does, the driver could potentially achieve better performance by mapping data buffers as noncacheable, so that it can avoid having to perform cache synchronization operations.

cache_invalidate_rate

Provides a runtime indication to the driver of the cost of invalidating the cache. The defined values for this field are similar to those defined for the *cache_flush_rate* field.

fd

The driver should set this field to NOFD.

Other fields in the structure should not be referenced or modified.

Cache coherency

Device drivers for hardware that performs Direct Memory Access (DMA) use this cache coherency library. These devices are either bus-mastering devices that can directly read or modify memory, or devices that use a DMA controller to transfer data between the device and memory. The key factor is that memory may be accessed by an agent other than the CPU(s).

On some systems, cache coherency is performed by a bus-snooping protocol that is implemented in hardware. In such systems, the CPU(s) snoop all transactions on the memory bus and automatically keep their caches in sync with the memory.

For example, if an external agent modifies data at a given memory location, the CPU will observe the memory write cycle, and if it has a cached copy of the data at that memory location, it will invalidate the corresponding cache entry. The next time the CPU tries to examine the memory location, it will fetch the modified data from memory, instead of retrieving stale data from the cache.

Similarly, special action is taken if an external agent attempts to read a memory location, and a CPU has modified the memory location, but the modified copy of the data is in its cache, and hasn't yet been written to memory. In this case, the read cycle is suspended while the CPU copies the updated version of the data out to memory. Once memory has been updated with the modified version, the read cycle can continue, and the external agent gets the updated copy of the data.

On other systems, where there is no such snooping protocol implemented in hardware, cache coherency between the CPU and external devices must be maintained by driver software. These are typically single-CPU systems, since on SMP systems, bus-snooping is the usual mechanism of keeping the CPUs in sync with each other. To work on these systems, special action needs to be taken by driver software, to ensure data coherency between the CPU and external devices.

A driver ensures data coherency for systems that don't have a bus-snooping protocol in different ways. The first one is the “big hammer” approach that simply disables caching for any memory location that can be accessed by both the CPU and the external device. This approach, however, has a severe performance penalty; for some network devices on certain systems, the throughput reduces to roughly half of the original value.

You can solve the above throughput problem by using cacheable data buffers, but perform synchronization operations on the data buffers at strategic points in the driver. For example, in the case of packet transmission for a network device, the driver must ensure that any data pertaining to the packet had been flushed out to memory, before allowing the DMA agent to begin copying the data. In the case of packet reception, the driver should invalidate any cached data pertaining to the packet buffer, before letting the DMA agent transfer data into the receive buffer. This eliminates the possibility that the CPU could write a cached portion of the data out to the memory buffer, after the network device had updated the buffer with new packet data.

The driver can perform cache flushing and invalidation operations in one of two ways. It can issue special CPU-dependent instructions that operate on the cache, or it can use the cache-coherency library (`libcache`). The latter approach is preferable, since it makes your driver portable. The library performs the correct thing based on the type of CPU it's running on. For maximum portability, the library must be used whether the system has a bus-snooping protocol or not. If the system implements a bus-snooping protocol, the library determines this, and ensures that there are no unnecessary synchronization operations being performed.

Returns:

0

Success.

-1

Failure and [errno](#) (p. 539) is set. If this function fails, it isn't safe for devices to DMA to or from cacheable buffers. Additionally, calling other functions with the `cache_ctl` structure that was provided will have unpredictable results.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The cache-invalidation operation could have certain negative side effects, which the driver must take measures to avoid. If a data buffer partially shares a cache line with some other piece of data (including another data buffer), data corruption could occur. Since the invalidation is performed with cache-line granularity, invalidating data at the start or end of the buffer could potentially invalidate important data, such as program variables, which means that changes made to the data by the CPU could be inadvertently lost.

You can avoid this by padding the data buffers. The driver should pad the start of the buffer, so that it starts on the next cache line boundary. It should also pad the buffer out to the end of the last cache line of the buffer. To do this, the driver can use the *cache_line_size* field of the *cache_ctl* structure. Note that this value could be zero (e.g. if there is a cache-coherency protocol implemented in hardware), in which case the driver doesn't need to do any padding.

CACHE_INVALID()

Invalidate cache lines associated with a data buffer

Synopsis:

```
#include <sys/cache.h>

CACHE_INVALID(struct cache_ctrl *cinfo,
               void *vaddr,
               uint64_t paddr,
               size_t len);
```

Arguments:

cinfo

A pointer to the structure that was initially passed to *cache_init()*.

vaddr

The virtual address of the buffer; this is a pointer to the data in the driver's virtual address space.

paddr

The physical address of the buffer: this is typically in the same address space that the external device will use to reference the data. The physical address is obtained by calling *mem_offset64()*. Since this function is fairly costly, drivers typically allocate a pool of data buffers at initialization (e.g. by calling *mmap()* with the MAP_PHYS and MAP_ANON flags), and pre-determine the physical addresses of the data.

len

The number of bytes in the buffer, for which the associated cache lines should be invalidated.

Library:

libcache

Use the -l cache option to qcc to link against this library.

Description:

This macro is used to invalidate any cache lines associated with a data buffer. This routine ensures that any subsequent modifications that are made to the data by an

external device will be not be corrupted by the CPU writing back cached data to the memory, and ensures that once the data has been modified, the CPU will fetch the updated data from memory, instead of retrieving stale data from the cache.



This macro was added in QNX Momentics 6.3.0 SP2.

Before using the `CACHE_*` macros on ARM platforms, the calling thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- obtain I/O privileges by calling [ThreadCtl\(\)](#) (p. 3408), specifying the `_NTO_TCTL_IO` flag:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

Failing to do so may result in a `SIGILL` (illegal instruction) signal.

Environment variables:

The following environment variables, if they exist, affect the behavior of this function:

`CACHE_NOP`

Instructs the library that the [CACHE_FLUSH\(\)](#) (p. 246) and [CACHE_INVALID\(\)](#) (p. 253) macros should have no effect.

`CACHE_MSINC`

Instructs the library that the [CACHE_FLUSH\(\)](#) (p. 246) and [CACHE_INVALID\(\)](#) (p. 253) macros should use the [msync\(\)](#) (p. 1825) C library call to perform cache synchronization.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

`CACHE_INVALID()` is implemented as a macro.

calloc()

Allocate space for an array

Synopsis:

```
#include <stdlib.h>

void* calloc ( size_t n,
               size_t size );
```

Arguments:

n

The number of array elements to allocate.

size

The size, in bytes, of one array element.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `calloc()` function allocates space from the heap for an array of *n* objects, each of *size* bytes, and initializes them to 0. Use [free\(\)](#) (p. 717) or [realloc\(\)](#) (p. 2527) to free the block of memory.



Because the `malloc()` implementation uses signed, 32-bit integers to represent the size internally, you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, `calloc()` indicates an error of `ENOMEM`.

If *n* or *size* is zero, the default behavior is to return a non-NULL pointer that's valid only to a corresponding call to `free()` or `realloc()`. Don't assume that this pointer points to any valid memory. You can control this behavior via the **`MALLOC_OPTIONS`** environmental variable; if the value of **`MALLOC_OPTIONS`** contains a *V*, `calloc()` returns a NULL pointer. This environment variable also affects [malloc\(\)](#) (p. 1569) and `realloc()`. This is known as the “System V” behavior.

Returns:

A pointer to the start of the allocated memory, or NULL if an error occurred (*errno* (p. 539) is set).

Errors:**ENOMEM**

Not enough memory.

EOK

No error.

Examples:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    char* buffer;

    buffer = (char* )calloc( 80, sizeof(char) );
    if( buffer == NULL ) {
        printf( "Can't allocate memory for buffer!\n" );
        return EXIT_FAILURE;
    }

    free( buffer );

    return EXIT_SUCCESS;
}
```

Environment variables:**MALLOC_OPTIONS**

Control the way *calloc()*, *malloc()*, and *realloc()* behave if you specify a size of 0 (or a value of 0 for the *n* argument to *calloc()*). The V (“System V”) and R (“use the *realloc()* behavior of QNX Neutrino 6.4.0 and earlier”) columns below indicate how the functions behave if the value of **MALLOC_OPTIONS** includes that letter:

Function	Default	V	R
<i>calloc</i> (<i>n</i> , 0)	Non-NULL	NULL	No effect
<i>malloc</i> (0)	Non-NULL	NULL	No effect
<i>realloc</i> (NULL, 0)	Non-NULL	NULL	No effect
<i>realloc</i> (non-NULL, 0)	Non-NULL	NULL	NULL

In all the above cases, if the function returns a non-NULL pointer, it's valid only for a corresponding call to *free()* or *realloc()*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

cbrt()*, *cbrtf()

Compute the cube root of a number

Synopsis:

```
#include <math.h>

double cbrt ( double x );
float cbrtf ( float x );
```

Arguments:

x

The number whose cube root you want to calculate.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *cbrt()* and *cbrtf()* functions compute the cube root of *x*.

Returns:

The cube root of *x*. If *x* is NAN, *cbrt()* returns NAN.

Examples:

```
#include <stdio.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv) {
    double a, b;

    a = 27.0;
    b = cbrt(a);
    printf("The cube root of %f is %f \n", a, b);

    return(0);
}
```

produces the output:

```
The cube root of 27.000000 is 3.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ceil(), ceilf()

Round up a value to the next integer

Synopsis:

```
#include <math.h>

double ceil( double x );
float ceilf( float x );
```

Arguments:

x

The value you want to round.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *ceil()* and *ceilf()* functions round the value of *x* up to the next integer (rounding towards the “ceiling”).

Returns:

The smallest integer $\geq x$.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f %f %f %f %f\n", ceil( -2.1 ),
            ceil( -2. ), ceil( 0.0 ), ceil( 2. ),
            ceil( 2.1 ) );
}
```



```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
-2.000000 -2.000000 0.000000 2.000000 3.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

cfgetispeed()

Return the input baud rate that's stored in a `termios` structure

Synopsis:

```
#include <termios.h>

speed_t cfgetispeed(
    const struct termios* termios_p );
```

Arguments:

termios_p

A pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *cfgetispeed()* function returns the input baud rate that's stored in the `termios` structure pointed to by *termios_p*.

You can get a valid `termios` control structure for an opened device by calling [tcgetattr\(\)](#) (p. 3347).

Returns:

The input baud rate stored in **termios_p*, or -1 if an error occurs (*errno* is set).

Errors:

EINVAL

One of the arguments is invalid.

ENOTTY

This function isn't supported by the system.

Examples:

```

#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "/dev/ser1", O_RDWR );
    tcgetattr( fd, &termios_p);

    /*
     * Get input baud rate
     */
    speed = cfgetispeed( &termios_p);
    printf( "Input baud: %ld\n", speed );

    close( fd );
    return EXIT_SUCCESS;
}

```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

cfgetospeed()

Return the output baud rate that's stored in a `termios` structure

Synopsis:

```
#include <termios.h>

speed_t cfgetospeed(
    const struct termios* termios_p );
```

Arguments:

termios_p

A pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *cfgetospeed()* function returns the output baud rate that's stored in the `termios` structure pointed to by *termios_p*.

You can get a valid `termios` control structure for an opened device by calling [tcgetattr\(\)](#) (p. 3347).

Returns:

The output baud rate stored in **termios_p*, or -1 if an error occurs (*errno* is set).

Errors:

EINVAL

One of the arguments is invalid.

ENOTTY

This function isn't supported by the system.

Examples:

```

#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "/dev/ser1", O_RDWR );
    tcgetattr( fd, &termios_p);

    /*
     * Get output baud rate
     */
    speed = cfgetospeed( &termios_p);
    printf( "Output baud: %ld\n", speed );

    close( fd );
    return EXIT_SUCCESS;
}

```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

cfgopen()

Open a configuration file

Synopsis:

```
#include <cfgopen.h>

int cfgopen( const char * path,
             unsigned flags,
             const char * historical,
             char * namebuf,
             int nblen );
```

Arguments:

path

The name of the configuration file that you want to open.

flags

Flags that control the opening; see below.

historical

A optional file to open as a last resort if none of the criteria for finding the path is met. This string works like a path search order, and lets you search more than one location. You can also specify %H to substitute the hostname value into the string. Specify NULL to ignore this option.

namebuf

A buffer to save the pathname in. Specify NULL to ignore this option.

nblen

The length of the buffer pointed to by *namebuf*. Specify 0 to ignore this option.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `cfgopen()` function opens the configuration file named by *path*. This function is a cover function for `open()` (p. 1905) that searches several default system locations for your files, based on specified characteristics.

The value of *flags* correspond to, and have similar limitations of, the standard `open()` flags. The *flags* value is constructed by the bitwise ORing of values from the following list, defined in the `<cfgopen.h>` header file. Applications must specify exactly one of these file-access modes in the value of *flag*:

CFGFILE_RDONLY

Open for reading only.

CFGFILE_RDWR

Open for reading and writing.

CFGFILE_WRONLY

Open for writing only.

You can also include any combination of these bits in the value of *flag*:

CFGFILE_APPEND

If set, the file offset is set to the end of the file prior to each write.

CFGFILE_CREAT

If the file doesn't exist, it's created with mode 0644, the file's user ID is set to the effective user ID of the process, and the group ID is set to the effective group ID of the process or the group ID of the file's parent directory (see `chmod()` (p. 289)).

CFGFILE_EXCL

If `CFGFILE_EXCL` and `CFGFILE_CREAT` are set, and the file exists, `cfgopen()` fails. The check for the existence of the file and the creation of the file if it doesn't exist is atomic with respect to other processes attempting the same operation with the same filename. Specifying `CFGFILE_EXCL` without `CFGFILE_CREAT` has no effect.

CFGFILE_TRUNC

If the file exists and is a regular file, and the file is successfully opened `CFGFILE_WRONLY` or `CFGFILE_RDWR`, the file length is truncated to zero and the mode and owner are left unchanged. `CFGFILE_TRUNC` has no effect on FIFO or block or character special files or directories. Using `CFGFILE_TRUNC` with `CFGFILE_RDONLY` has no effect.

Search condition flags

In order to hint to the function where it should access or construct (in the case of `CFGFILE_CREAT`) *path*, there are several bits that you can specify and OR into *flags*. When specified, the bits are accessed using the following search order:

1. `CFGFILE_USER_NODE`

\$HOME/.cfg/node_name/path

2. `CFGFILE_USER`

\$HOME/.cfg/path

3. `CFGFILE_NODE`

/etc/host_cfg/node_name/path

4. `CFGFILE_GLOBAL`

path

where *node_name* is the value you get by calling [confstr\(\)](#) (p. 344) for `CS_HOSTNAME`.

If the directory `/etc/host_cfg` doesn't exist on the system, the following *flags* are transformed automatically:



- `CFGFILE_USER_NODE` becomes `CFGFILE_USER`
- `CFGFILE_NODE` becomes `CFGFILE_GLOBAL`

When creating a file or opening a file for writing, you can specify only one of the above location flags. Set `CFGFILE_NOFD` when you need only the pathname, not the file descriptor. If a directory path doesn't exist when a file is opened for creation, *cfgopen()* attempts to create the path.

Returns:

A valid file descriptor if `CFGFILE_NOFD` isn't specified, a nonnegative value if `CFGFILE_NOFD` is specified, or -1 if an error occurs.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

cfmakeraw()

Set terminal attributes

Synopsis:

```
#include <termios.h>

int cfmakeraw( struct termios * termios_p );
```

Arguments:

termios_p

A pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *cfmakeraw()* function sets the terminal attributes as follows:

```
termios_p->c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL|IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO|ECHONL|ICANON|ISIG|IEXTEN);
termios_p->c_cflag &= ~(CSIZE|PARENB);
termios_p->c_cflag |= CS8;
```

You can get a valid `termios` control structure for an opened device by calling [tcgetattr\(\)](#) (p. 3347).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) indicates the reason).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

cfree()

Free allocated memory

Synopsis:

```
#include <malloc.h>

int cfree( void *ptr );
```

Arguments:

ptr

A pointer to the block of memory that you want to free. It's safe to call *cfree()* with a NULL pointer.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *cfree()* function deallocates the memory block specified by *ptr*, which was previously returned by a call to [calloc\(\)](#) (p. 255), [malloc\(\)](#) (p. 1569) or [realloc\(\)](#) (p. 2527).

Returns:

1

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Calling *cfree()* on a pointer already deallocated by a call to *cfree()*, *free()*, or *realloc()* could corrupt the memory allocator's data structures.

cfsetispeed()

Set the input baud rate in a `termios` structure

Synopsis:

```
#include <termios.h>

int cfsetispeed( struct termios* termios_p,
                 speed_t speed );
```

Arguments:

termios_p

A pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes.

speed

The new speed. Valid values for *speed* are defined in `<termios.h>`.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *cfsetispeed()* function sets the input baud rate within the `termios` structure pointed to by *termios_p* to be *speed*.

You can get a valid `termios` control structure for an opened device by calling [tcgetattr\(\)](#) (p. 3347).

-
- The new baud rate isn't effective until you call [tcsetattr\(\)](#) (p. 3364) with this modified `termios` structure.
 - Attempts to set baud rates to values that aren't supported by the hardware are ignored, and cause *tcsetattr()* to return an error, but *cfsetispeed()* doesn't indicate an error.
 - Attempts to set input baud rates to a value that's different from the output baud rate, when the hardware doesn't support split baud rates, cause the input baud rate to be ignored, but no error is generated.
-



Returns:

0

Success.

-1

An error occurred (*errno* is set).**Errors:****EINVAL**

One of the arguments is invalid.

ENOTTY

This function isn't supported by the system.

Examples:

```

#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "/dev/ser1", O_RDWR );
    tcgetattr( fd, &termios_p);

    /*
     *   Set input baud rate
     */
    speed = 9600;
    cfsetispeed( &termios_p, speed );
    tcsetattr( fd, TCSADRAIN, &termios_p);

    close( fd );
    return EXIT_SUCCESS;
}

```

Classification:[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

cfsetospeed()

Set the output baud rate in a `termios` structure

Synopsis:

```
#include <termios.h>

int cfsetospeed( struct termios *termios_p,
                  speed_t speed );
```

Arguments:

termios_p

A pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes.

speed

The new speed. Valid values for *speed* are defined in `<termios.h>`.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `cfsetospeed()` function sets the output baud rate within the `termios` structure pointed to by *termios_p* to be *speed*.

You can get a valid `termios` control structure for an opened device by calling [tcgetattr\(\)](#) (p. 3347).



- The new baud rate isn't effective until you call [tcsetattr\(\)](#) (p. 3364), with this modified `termios` structure.
- Attempts to set baud rates to values that aren't supported by the hardware are ignored, and cause `tcsetattr()` to return an error, but `cfsetospeed()` doesn't indicate an error.

Setting the output baud rate to `B0` causes the connection to be dropped. If *termios_p* represents a modem, the modem control lines will be turned off.

Returns:

0

Success.

-1

An error occurred (*errno* indicates the reason).**Errors:****EINVAL**

One of the arguments is invalid.

ENOTTY

This function isn't supported by the system.

Examples:

```

#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "/dev/ser1", O_RDWR );
    tcgetattr( fd, &termios_p);

    /*
     *   Set output baud rate
     */
    speed = B9600;
    cfsetospeed( &termios_p, speed );
    tcsetattr( fd, TCSADRAIN, &termios_p);

    close( fd );
    return EXIT_SUCCESS;
}

```

Classification:[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ChannelCreate(), ChannelCreate_r()

Create a communications channel

Synopsis:

```
#include <sys/neutrino.h>

int ChannelCreate( unsigned flags );
int ChannelCreate_r( unsigned flags );
```

Arguments:

flags

Flags that can be used to request notification pulses from the kernel or request other changes in behavior; a combination of the following:

- [_NTO_CHF_COID_DISCONNECT](#) (p. 280)
- [_NTO_CHF_DISCONNECT](#) (p. 280)
- [_NTO_CHF_FIXED_PRIORITY](#) (p. 281)
- [_NTO_CHF_NET_MSG](#) (p. 281)
- [_NTO_CHF_PRIVATE](#) (p. 281)
- [_NTO_CHF_REPLY_LEN](#) (p. 281)
- [_NTO_CHF_SENDER_LEN](#) (p. 281)
- [_NTO_CHF_THREAD_DEATH](#) (p. 281)
- [_NTO_CHF_UNBLOCK](#) (p. 282)

For more information, see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ChannelCreate()* and *ChannelCreate_r()* kernel calls create a channel that can be used to receive messages and pulses. Once created, the channel is owned by the process and isn't bound to the creating thread.

These functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 283) section for details.

Threads wishing to communicate with the channel attach to it by calling [ConnectAttach\(\)](#) (p. 351). The threads may be in the same process, or in another process on the same node (or a remote node if the network manager is running).



If a process wants other processes to communicate with it, it typically uses [name_attach\(\)](#) (p. 1842) to create a channel and associate a name with it, and the sender process uses [name_open\(\)](#) (p. 1852) to locate that name and create a connection to it.

Once attached, these threads use [MsgSendv\(\)](#) (p. 1801) or [MsgSendPulse\(\)](#) (p. 1789) to enqueue messages and pulses on the channel. Messages and pulses are enqueued in priority order.

To dequeue and read messages and pulses from a channel, use [MsgReceivev\(\)](#) (p. 1771). Any number of threads may call [MsgReceivev\(\)](#) at the same time, in which case they block and queue (if no messages or pulses are waiting) for a message or pulse to arrive. A multi-threaded I/O manager typically creates multiple threads and has them all RECEIVE-blocked on the channel.

The return value of [ChannelCreate\(\)](#) is a channel ID, an `int` taken from a channel vector on the process. Most managers use a single channel for most, if not all, their communications with clients. Additional channels can be used as special channels for information.

By default, when a message is received from a channel, the thread priority of the receiver is set to match that of the thread that sent the message. This basic priority inheritance prevents priority inversion. If a message arrives at a channel and there's no thread waiting to receive it, the system boosts (if necessary) all threads in the process that have received a message from the channel in the past. This boost prevents a priority inversion of the client in the case where all threads are currently working on behalf of other clients, perhaps at a lower priority. When a thread is first created, it isn't associated with a channel until it does a [MsgReceivev\(\)](#) on it. In the case of multiple channels, a thread is associated with the last channel it received from.

After receiving a message, a thread can dissociate itself from the channel by calling [MsgReceivev\(\)](#) with a -1 for the channel ID. Priority inheritance can be disabled by setting `_NTO_CHF_FIXED_PRIORITY` in the *flags* argument. In this case a thread's priority isn't affected by messages it receives on a channel.

A manager typically involves the following loop. There may be one or more threads in the loop at a time. Typically your program calls [ChannelCreate\(\)](#) only once, and all threads block on that channel.

```
iov_t iov;
...
SETIOV( &iov, &msg, sizeof( msg ) );
chid = ChannelCreate(flags);
...
for(;;) {
```

```

    /*
    Here's a one-part message; you could just as
    easily receive a 20-part message by filling in the
    iov appropriately.
    */
    rcvid = MsgReceivev(chid, &iov, 1, &info);

    /* msg is filled in by MsgReceivev() */
    switch(msg.type) {
        ...
    }

    /* iov could be filled in again to point to a new message */
    MsgReplyv(rcvid, status, iov, 1);
}

```

Some of the channel flags in the *flags* argument request changes from the default behavior; others request notification pulses from the kernel. The pulses are received by *MsgReceivev()* on the channel and are described by a [_pulse](#) (p. 2440) structure.

The channel flags and (where appropriate) associated values for the pulse's *code* and *value* are described below.

`_NTO_CHF_COID_DISCONNECT`

Pulse code:

`_PULSE_CODE_COIDDEATH`

Pulse value:

Connection ID (*coid*) of a connection that was attached to a destroyed channel.

Deliver a pulse to this channel for each connection that belongs to the calling process when the channel that the connection is attached to is destroyed. Only one channel per process can have this flag set.

If a server exits or closes its channel at more or less the same time that the client closes a connection to the channel, the kernel might or might not send a `_PULSE_CODE_COIDDEATH` pulse to the client. If the client then opens a new connection to another server before getting the pulse, the pulse will seem to indicate that it's the new server that has died. Your code for handling the `_PULSE_CODE_COIDDEATH` pulse needs to include something like this:

```

void
got_pulse(struct _pulse *pulse) {
    if(pulse->type == _PULSE_CODE_COIDDEATH) {
        int coid = pulse->value.sival_int;

        if(ConnectServerInfo(0, coid, NULL) != coid) {
            // server's really gone, so clean up the connection state
        } else {
            // stale pulse; probably can ignore it
        }
    }
}

```

`_NTO_CHF_DISCONNECT`

Pulse code:

_PULSE_CODE_DISCONNECT**Pulse value:**

None

Deliver a pulse when all connections from a process are detached (e.g., [close\(\)](#) (p. 336), [ConnectDetach\(\)](#) (p. 367), [name_close\(\)](#) (p. 1848)). If a process dies without detaching all its connections, the kernel detaches them from it. When this flag is set, the server must call *ConnectDetach(scoid)* where *scoid* is the server connection ID in the pulse message. Failure to do so leaves an invalid server connection ID that can't be reused. Over time, the server may run out of available IDs. If this flag isn't set, the kernel removes the server connection ID automatically, making it available for reuse.

_NTO_CHF_FIXED_PRIORITY

Suppress priority inheritance when receiving messages. Receiving threads won't change their priorities to those of the sending threads. If you're using adaptive partitioning, the receiving threads won't run in the sending threads' partitions.

_NTO_CHF_NET_MSG

Reserved for the `io_pkt*` resource manager.

_NTO_CHF_PRIVATE

Mark the channel as being private. There are some cases where it's useful to have a channel for a process's internal use, but you don't want other people to be able to attach to it. If another process calls [ConnectAttach\(\)](#) (p. 351) to attach to your process's private channel, that call fails with an error code of `EPERM`.

In order to create a public channel (i.e., *without* `_NTO_CHF_PRIVATE` set, your process must have the `PROCmgr_AID_PUBLIC_CHANNEL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

_NTO_CHF_REPLY_LEN

Request that the length of the reply be included in the *dstmsglen* member of the [_msg_info](#) (p. 1731) structure that [MsgReceivev\(\)](#) (p. 1771) fills in. The *dstmsglen* member is valid only if you set this channel flag when you create the channel.

_NTO_CHF_SENDER_LEN

Request that the length of the source message be included in the *srcmsglen* member of the [_msg_info](#) (p. 1731), structure that [MsgReceivev\(\)](#) (p. 1771) fills in. The *srcmsglen* member is valid only if you set this channel flag when you create the channel.

_NTO_CHF_THREAD_DEATH**Pulse code:**`_PULSE_CODE_THREADDEATH`

Pulse value:Thread ID (*tid*)

Deliver a pulse on the death of any thread in the process that owns the channel. Only one channel per process can have this flag set.

_NTO_CHF_UNBLOCK**Pulse code:**`_PULSE_CODE_UNBLOCK`**Pulse value:**Receive ID (*rcvid*)

In most cases, you'll set the `_NTO_CHF_UNBLOCK` flag.

Deliver a pulse when a thread that's REPLY-blocked on a channel attempts to unblock before its message is replied to. This occurs between the time of a [MsgReceivev\(\)](#) (p. 1771) and a [MsgReplyv\(\)](#) (p. 1778) by the server. The sending thread may be unblocked because of a signal or a kernel timeout.

If the sending thread unblocks, [MsgReplyv\(\)](#) fails. The manager may not be in a position to handle this failure. It's also possible that the client will die because of the signal and never send another message. If the manager is holding onto resources for the client (such as an open file), it may want to receive notification that the client wants to break out of its [MsgSendv\(\)](#) (p. 1801).

Setting the `_NTO_CHF_UNBLOCK` bit in *flags* prevents a thread that's in the REPLY-blocked state from unblocking. Instead, a pulse is sent to the channel, informing the manager that the client wishes to unblock. In the case of a signal, the signal will be pending on the client thread. When the manager replies, the client is unblocked and at that point, any pending signals are acted upon. From the client's point of view, its [MsgSendv\(\)](#) will have completed normally and any signal will have arrived on the opcode following the successful kernel call.

When the manager receives the pulse, it can do one of these things:

- If it believes that it will be replying shortly, it can discard the pulse, resulting in a small latency in the unblocking, or it can signal the client. A short blocking request to a filesystem often takes this approach.
- If the reply is going to take some time or an unknown amount of time, the manager should cancel the current operation and reply back with an error or whatever data is available at this time in the reply message to the client thread. A request to a device manager waiting for input would take this approach.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ChannelCreate()

The channel ID of the newly created channel. If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

ChannelCreate_r()

The channel ID of the newly created channel. This function does **NOT** set *errno*. If an error occurs, the function returns the negative of a value from the Errors section.

Errors:**EAGAIN**

All kernel channel objects are in use.

EPERM

The process tried to create a public channel without having the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ChannelDestroy(), ChannelDestroy_r()

Destroy a communications channel

Synopsis:

```
#include <sys/neutrino.h>

int ChannelDestroy( int chid );
int ChannelDestroy_r( int chid );
```

Arguments:

chid

The channel ID, returned by [ChannelCreate\(\)](#) (p. 278), of the channel that you want to destroy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls remove a channel specified by the channel ID *chid* argument. Once destroyed, any attempt to receive messages or pulses on the channel will fail. Any threads that are blocked on the channel by calling [MsgReceivev\(\)](#) (p. 1771) or [MsgSendv\(\)](#) (p. 1801) will be unblocked and return with an error.

The *ChannelDestroy()* and *ChannelDestroy_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 285) section for details.

When the channel is destroyed, all server connection IDs become invalid. The client connections are also marked invalid but remain in existence until the client removes them by calling [ConnectDetach\(\)](#) (p. 367). An attempt by the client to use one of these invalid connections using *MsgSendv()* or [MsgSendPulse\(\)](#) (p. 1789) will return with an error.

A server typically destroys its channels prior to its termination. If it fails to do so, the kernel destroys them automatically when the process dies.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ChannelDestroy()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

ChannelDestroy_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, the function may return any value in the Errors section.

Errors:**EINVAL**

The channel specified by *chid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

chdir()

Change the current working directory

Synopsis:

```
#include <unistd.h>

int chdir( const char* path );
```

Arguments:

path

The new current working directory.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *chdir()* function changes the current working directory to *path*, which can be relative to the current working directory or an absolute path name.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission is denied for a component of *path*.

EINTR

The call was interrupted by a signal.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The *path* argument is longer than PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT

The specified *path* doesn't exist, or *path* is an empty string.

ENOMEM

There wasn't enough memory to allocate a control structure.

ENOSYS

The *chdir()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of *path* is not a directory.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char* argv[] )
{
    if( argc != 2 ) {
        fprintf( stderr, "Use: cd <directory>\n" );
        return EXIT_FAILURE;
    }

    if( chdir( argv[1] ) == 0 ) {
        printf( "Directory changed to %s\n", argv[1] );
        return EXIT_SUCCESS;
    } else {
        perror( argv[1] );
        return EXIT_FAILURE;
    }
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

There's only one current working directory per *process*. In a multithreaded application, any thread calling *chdir()* will change the current working directory for all threads in that process.

chmod()

Change the permissions for a file

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod( const char * path,
           mode_t mode );
```

Arguments:

path

The name of the file whose permissions you want to change.

mode

The new permissions for the file. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *chmod()* function changes `S_ISUID`, `S_ISGID`, `S_ISVTX` and the file permission bits of the file specified by the pathname pointed to by *path* to the corresponding bits in the *mode* argument. The application must ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges to do this.

If a directory is writable and the sticky bit (`S_ISVTX`) is set on the directory, a process can remove or rename a file within that directory only if one or more of the following is also true:

- The effective user ID of the process matches the file's owner ID.
- The effective user ID of the process matches the directory's owner ID.
- The file is writable by the effective user ID of the process.
- The user is a superuser (effective user ID of 0).

If a directory has the set-group ID bit set, a file created in that directory will have the same group ID as that directory. Otherwise, the newly created file's group ID is set to the effective group ID of the creating process.

If the calling process doesn't have appropriate privileges, and if the group ID of the file doesn't match the effective group ID, and the file is a regular file, bit `S_ISGID` (set-group-ID on execution) in the file's mode is cleared on a successful return from `chmod()`.

If the effective user ID of the calling process is equal to the file owner, or the calling process has appropriate privileges (for example, it belongs to the superuser), `chmod()` sets `S_ISUID`, `S_ISGID` and the file permission bits, defined in the `<sys/stat.h>` header file, from the corresponding bits in the *mode* argument. These bits define access permissions for the user associated with the file, the group associated with the file and all others.

This call has no effect on file descriptors for files that are already open.

If `chmod()` succeeds, the `st_ctime` field of the file is marked for update.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission is denied on a component of the path prefix.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of the *path* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENOTDIR

A component of the path prefix isn't a directory.

ENOENT

The file doesn't exist, or the *path* arguments points to an empty string.

ENOSYS

The *chmod()* function isn't implemented for the filesystem specified in *path*.

EPERM

The effective user ID doesn't match the owner of the file, and the calling process doesn't have appropriate privileges.

EROFS

The file resides on a read-only filesystem.

Examples:

```
/*
 * Change the permissions of a list of files
 * to be read/write by the owner only
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( chmod( argv[i], S_IRUSR | S_IWUSR ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }

    return ecode;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

chown()

Change the user ID and group ID of a file

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

int chown( const char * path,
           uid_t owner,
           gid_t group );
```

Arguments:

path

The name of the file whose ownership you want to change.

owner

The user ID of the new owner.

group

The group ID of the new owner.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *chown()* function changes the user ID and group ID of the file specified by *path* to be the numeric values contained in *owner* and *group*, respectively.

If the named file is a symbolic link, *chown()* changes the ownership of the file or directory to which the symbolic link refers; *lchown()* (p. 1502) changes the ownership of the symbolic link file itself.

Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges (for example, the superuser) may change the ownership of a file.

In QNX Neutrino, the `_POSIX_CHOWN_RESTRICTED` flag (tested via the `_PC_CHOWN_RESTRICTED` flag in *pathconf()* (p. 1936)), is enforced for *path*. This means

that only the superuser may change the ownership or the group of a file to anyone. Normal users can't give a file away to another user by changing the file ownership, nor to another group by changing the group ownership.

If the *path* argument refers to a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode are cleared, if the function is successful.

If *chown()* succeeds, the *st_ctime* field of the file is marked for update.

Returns:

0

Success.

-1 (no changes were made in the user ID and group ID of the file).

An error occurred (*errno* (p. 539) is set).

Errors:

EACCES

Search permission is denied on a component of the path prefix.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of the *path* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT

A component of the path prefix doesn't exist, or the *path* arguments points to an empty string.

ENOSYS

The *chown()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix isn't a directory.

EPERM

The effective user ID doesn't match the owner of the file, or the calling process doesn't have appropriate privileges.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
/*
 * Change the ownership of a list of files
 * to the current user/group
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( int argc, char** argv )
{
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( chown( argv[i], getuid(), getgid() ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }
    exit( ecode );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

chroot()

Change the root directory

Synopsis:

```
#include <unistd.h>

int chroot( const char *path );
```

Arguments:

path

The name of the new root directory. This can include a network root (e.g. */net/node_name*).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *chroot()* function causes the *path* directory to become the *root directory*, the starting point for path searches for path names beginning with */*. The user's working directory is unaffected.

The `..` entry in the root directory is interpreted to mean the root directory itself. Thus, you can't use `..` to access files outside the subtree rooted at the root directory.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EACCES

Search permission is denied for a component of *path*.

EBADF

The descriptor isn't valid.

EFAULT

The *path* argument points to an illegal address.

EINTR

A signal was caught during the *chroot()* function.

EIO

An I/O error occurred while reading from or writing to the filesystem.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of *path* require hopping to multiple remote machines, and the filesystem type doesn't allow it.

ENAMETOOLONG

The length of the *path* argument exceeds **{PATH_MAX}**, or the length of a path component exceeds **{NAME_MAX}** while **{_POSIX_NO_TRUNC}** is in effect.

ENOENT

The named directory doesn't exist or is a null pathname.

ENOLINK

The *path* points to a remote machine and the link to that machine is no longer active.

ENOTDIR

Any component of the path name isn't a directory.

EPERM

The effective user of the calling process isn't the superuser.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

chsize()

Change the size of a file

Synopsis:

```
#include <unistd.h>

int chsize( int filedes,
            long size );
```

Arguments:

filedes

A file descriptor for the file whose size you want to change.

size

The new size of the file, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *chsize()* function extends or truncates the file specified by *filedes* to *size* bytes. The file is padded with NUL (`'\0'`) characters if it needs to be extended.



The *chsize()* function ignores advisory locks that may have been set with the [fcntl\(\)](#) (p. 623) function.

Returns:

0

Success.

-1

An error occurred.

Errors:

EBADF

The *filedes* argument isn't a valid file descriptor, or the file isn't opened for writing.

ENOSPC

There isn't enough space left on the device to extend the file.

ENOSYS

The *chsize()* function isn't implemented for the filesystem specified by *filedes*.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main( void )
{
    int filedес;

    filedес= open( "file", O_RDWR | O_CREAT,
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
    if( filedес!= -1 ) {
        if( chsize( filedес, 32 * 1024L ) != 0 ) {
            printf( "Error extending file\n" );
        }
        close( filedес);
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

clearenv()

Clear the environment

Synopsis:

```
#include <stdlib.h>

int clearenv( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *clearenv()* function clears the environment area; no environment variables are defined immediately after the *clearenv()* call.

Note that *clearenv()* clears the following environment variables, which may then affect the operation of other library functions such as [spawnp\(\)](#) (p. 3122):

- ***PATH***
- ***SHELL***
- ***TERM***
- ***TERMINFO***
- ***LINES***
- ***COLUMNS***
- ***TZ***

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ENOMEM

Not enough memory to allocate a control structure.

Examples:

Clear the entire environment and set up a new **TZ** environment variable:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( clearenv() != 0 ) {
        puts( "Unable to clear the environment" );
        return EXIT_FAILURE;
    }

    setenv( "TZ", "EST5EDT", 0 );

    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The `clearenv()` function manipulates the environment pointed to by the global [environ](#) (p. 528) variable.

clearerr()

Clear a stream's end-of-file and error flags

Synopsis:

```
#include <stdio.h>

void clearerr( FILE *fp );
```

Arguments:

fp

The stream for which you want to clear the flags.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *clearerr()* function clears the end-of-file and error flags for the stream specified by *fp*.

These indicators are also cleared when the file is opened, or by an explicit call to *clearerr()* or [rewind\(\)](#) (p. 2646).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    c = 'J';
    fp = fopen( "file", "w" );
    if( fp != NULL ) {
        fputc( c, fp );
        if( ferror( fp ) ) { /* if error          */
            clearerr( fp ); /* clear the error */
            fputc( c, fp ); /* and retry it    */
        }
    }

    fclose( fp );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ClientInfoExtFree()

Free the structure used for extended information about a client connection

Synopsis:

```
#include <sys/neutrino.h>

int ClientInfoExtFree( struct _client_info ** info );
```

Arguments:

info

The address of a pointer to the `_client_info` structure that you want to free.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This call frees the memory allocated by a call to [ConnectClientInfoExt\(\)](#) (p. 363).

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

clock()

Return the number of clock ticks used by the program

Synopsis:

```
#include <time.h>

clock_t clock( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `clock()` function returns the number of clock ticks of processor time used by the program since it started executing. You can convert the number of ticks into seconds by dividing by the value `CLOCKS_PER_SEC`.



In a multithreaded program, `clock()` returns the time used by *all* threads in the application; `clock()` returns the time since the program started, not the time since a specific thread started.

Returns:

The number of clock ticks, or `(clock_t) -1` if the number of ticks couldn't be determined or exceeds the maximum value that the `clock_t` type can represent.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

void compute( void )
{
    int i, j;
    double x;

    x = 0.0;
    for( i = 1; i <= 100; i++ ) {
        for( j = 1; j <= 100; j++ ) {
            x += sqrt( (double) i * j );
        }
    }

    printf( "%16.7f\n", x );
}
```

```
int main( void )
{
    clock_t start_time, end_time;

    start_time = clock();
    compute();
    end_time = clock();
    printf( "Execution time was %lu seconds\n",
           (long) ((end_time - start_time) / CLOCKS_PER_SEC) );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

clock_getcpuclockid()

Return the clock ID of the CPU-time clock from a specified process

Synopsis:

```
#include <sys/types.h>
#include <time.h>
#include <errno.h>

int clock_getcpuclockid( pid_t pid,
                        clockid_t* clock_id );
```

Arguments:

pid

The process ID for the process whose clock ID you want to get.

clock_id

A pointer to a `clockid_t` object where the function can store the clock ID.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `clock_getcpuclockid()` function gets the clock ID of the CPU-time clock of the process specified by *pid* and stores it in the object pointed to by *clock_id*. The CPU-time clock represents the amount of time the process has spent actually running.

If *pid* is zero, the clock ID of the CPU-time clock of the process marking the call is returned in *clock_id*.

A process always has permission to obtain the CPU-time clock ID of another process.

Returns:

0

Success.

ESRCH

No process can be found corresponding to the specified *pid*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

int main( int argc, const char *argv[] )
{
    clockid_t clk_id;
    struct timespec tspec;
    int pid;

    /* Get the amount of time that the kernel has spent running. */

    pid = 1;

    if (clock_getcpuclockid( pid, &clk_id) == 0)
    {
        if (clock_gettime( clk_id, &tspec ) != 0)
        {
            perror ("clock_gettime()");
        }
        else
        {
            printf ("CPU time for pid %d is %d seconds, %ld nanoseconds.\n",
                    pid, tspec.tv_sec, tspec.tv_nsec);
        }
    }
    else
    {
        printf ("clock_getcpuclockid(): no process with ID %d.\n", pid);
    }

    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1 CPT*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

clock_getres()

Get the resolution of the clock

Synopsis:

```
#include <time.h>

int clock_getres( clockid_t clock_id,
                  struct timespec * res );
```

Arguments:

clock_id

The ID of the clock whose resolution you want to get; one of:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

res

A pointer to a [timespec](#) (p. 3473) structure in which *clock_getres()* can store the resolution. The function sets the *tv_sec* member to 0, and the *tv_nsec* member to be the resolution of the clock, in nanoseconds.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *clock_getres()* function gets the resolution of the clock specified by *clock_id* and puts it into the buffer pointed to by *res*.

Returns:

- 0
Success
- 1
An error occurred (*errno* (p. 539) is set).

Errors:

- EFAULT**
A fault occurred trying to access the buffers provided.
- EINVAL**
Invalid *clock_id*.

Examples:

```
/*
 * This program prints out the clock resolution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    struct timespec res;

    if ( clock_getres( CLOCK_REALTIME, &res) == -1 ) {
        perror( "clock get resolution" );
        return EXIT_FAILURE;
    }
    printf( "Resolution is %ld micro seconds.\n",
           res.tv_nsec / 1000 );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 TMR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

clock_gettime()

Get the current time of a clock

Synopsis:

```
#include <time.h>

int clock_gettime( clockid_t clock_id,
                  struct timespec * tp );
```

Arguments:

clock_id

The ID of the clock whose time you want to get; one of:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.
- A clock ID returned by [clock_getcpuclockid\(\)](#) (p. 307) or [pthread_getcpuclockid\(\)](#) (p. 2295), representing the amount of time the process or thread has spent actually running.

tp

A pointer to a [timespec](#) (p. 3473) structure where *clock_gettime()* can store the time. This function sets the members as follows:

- *tv_sec* — the number of seconds since 1970.
- *tv_nsec* — the number of nanoseconds expired in the current second. This value increases by some multiple of nanoseconds, based on the system clock's resolution.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `clock_gettime()` function gets the current time of the clock specified by `clock_id`, and puts it into the buffer pointed to by `tp`.

Returns:

0

Success.

-1

An error occurred (`errno` is set).

Errors:

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

Invalid `clock_id`.

ESRCH

The process associated with this request doesn't exist.

Examples:

```
/*
 * This program calculates the time required to
 * execute the program specified as its first argument.
 * The time is printed in seconds, on standard out.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#define BILLION 1000000000L;

int main( int argc, char** argv )
{
    struct timespec start, stop;
    double accum;

    if( clock_gettime( CLOCK_REALTIME, &start) == -1 ) {
        perror( "clock_gettime" );
        return EXIT_FAILURE;
    }
}
```

```
system( argv[1] );

if( clock_gettime( CLOCK_REALTIME, &stop) == -1 ) {
    perror( "clock_gettime" );
    return EXIT_FAILURE;
}

accum = ( stop.tv_sec - start.tv_sec )
        + (double)( stop.tv_nsec - start.tv_nsec )
          / (double)BILLION;
printf( "%lf\n", accum );
return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 TMR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

clock_nanosleep()

High resolution sleep with specifiable clock

Synopsis:

```
#include <time.h>

int clock_nanosleep( clockid_t clock_id,
                    int flags,
                    const struct timespec * rntp,
                    struct timespec * rmtp );
```

Arguments:

clock_id

The ID of the clock to use to measure the time. The possible clock types are:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

The *clock_nanosleep()* function fails if the *clock_id* argument refers to the CPU-time clock of the calling thread.

flags

Flags that specify when the current thread is to be suspended from execution:

- when the time interval specified by the *rntp* argument has elapsed (**TIMER_ABSTIME** is *not* set).
- when the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rntp* argument (**TIMER_ABSTIME** is set).

If, at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then *clock_nanosleep()* returns immediately, and the calling process isn't suspended.

- when a signal is delivered to the calling thread, and the signal's action is to invoke a signal-catching function or terminate the process.

Calling *clock_nanosleep()* with *TIMER_ABSTIME* *not* set, and *clock_id* set to *CLOCK_REALTIME* is the equivalent to calling *nanosleep()* (p. 1855) with the same *rqtp* and *rmtp* arguments.

rqtp

A pointer to a *timespec* (p. 3473) structure that specifies the time interval between the requested time and the time actually slept.

rmtp

NULL, or a pointer to a *timespec* in which the function can store the amount of time remaining in an interval.

For the relative *clock_nanosleep()* function, if *rmtp* isn't NULL, the *timespec* structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If it's NULL, the remaining time isn't returned.

The absolute *clock_nanosleep()* function has no effect on the structure referenced by *rmtp*.

Library:

libc

Use the *-l c* option to *gcc* to link against this library. This library is usually included automatically.

Description:

The *clock_nanosleep()* function suspends the current thread from execution until:

- If *TIMER_ABSTIME* is set, the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rqtp* argument.

Or:

- If *TIMER_ABSTIME* is *not* set, the time interval specified by the *rqtp* argument has elapsed.

Or:

- A signal is delivered to the calling thread, and the signal's action is to invoke a signal-catching function or terminate the process.

The [nanosleep\(\)](#) (p. 1855) function always uses CLOCK_REALTIME.

The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution (see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*) or because of scheduling and other system activity. Except for the case of being interrupted by a signal, the suspension time for:

- the relative *clock_nanosleep()* function (TIMER_ABSTIME not set) — isn't less than the time interval specified by *rqtp*, as measured by the corresponding clock
- the absolute *clock_nanosleep()* function (TIMER_ABSTIME set) — is in effect at least until the value of the corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being interrupted by a signal.

Using the *clock_nanosleep()* function has no effect on the action or blockage of any signal.

Returns:

Zero if the requested time has elapsed, or a corresponding error value if *clock_nanosleep()* has been interrupted by a signal, or fails.

Errors:

EINTR

The call was interrupted by a signal.

EINVAL

The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million; or TIMER_ABSTIME is specified in *flags* and the *rqtp* argument is outside the range for the clock specified by *clock_id*; or the *clock_id* argument doesn't specify a known clock, or specifies the CPU-time clock of the calling thread.

ENOTSUP

The *clock_id* argument specifies a clock for which *clock_nanosleep()* isn't supported, such as a CPU-time clock.

Classification:

[POSIX 1003.1 CS](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

clock_settime()

Set a clock

Synopsis:

```
#include <time.h>

int clock_settime( clockid_t id,
                  const struct timespec * tp );
```

Arguments:

id

The clock ID, `CLOCK_REALTIME` or `CLOCK_MONOTONIC`, that maintains the system time, or the clock ID that's returned by [ClockId\(\)](#) (p. 327).

tp

A pointer to a [timespec](#) (p. 3473) structure containing at least the following members:

- *tv_sec* — the number of seconds since 1970.
- *tv_nsec* — the number of nanoseconds in the current second. This value increases by some multiple of nanoseconds, based on the system clock's resolution.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `clock_settime()` function sets the clock specified by *id* to the time specified in the buffer pointed to by *tp*.



- Be careful if you set the date during the period that a time zone is switching from daylight saving time (DST) to standard time. When a time zone changes to standard time, the local time goes back one hour (for example, 2:00 a.m. becomes 1:00 a.m.). The local time during this hour is ambiguous (e.g. 1:14 a.m. occurs twice in the morning that the time zone switches to

standard time). To avoid problems, use UTC time to set the date in this period.

- You can't set the time when the *id* is `CLOCK_MONOTONIC`.
 - In order to set the clock, your process must have the `PROCmgr_AID_CLOCKSET` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
-

Returns:

0

Success

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

One of the following occurred:

- The *id* is invalid.
- The number of nanoseconds specified by the *tv_nsec* member is less than zero or greater than or equal to 1000 million.
- The *tv_sec* member is -1 (which could happen if you set it to the result from [mktime\(\)](#) (p. 1649) without checking to see if the call succeeded).

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

```
/* This program sets the clock forward 1 day. */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

int main( void )
{
    struct timespec stime;

    if( clock_gettime( CLOCK_REALTIME, &stime) == -1 ) {
        perror( "getclock" );
        return EXIT_FAILURE;
    }

    stime.tv_sec += (60*60)*24L; /* Add one day */
    stime.tv_nsec = 0;
```

```
if( clock_settime( CLOCK_REALTIME, &stime) == -1 ) {  
    perror( "setclock" );  
    return EXIT_FAILURE;  
}  
return EXIT_SUCCESS;  
}
```

Classification:

POSIX 1003.1 TMR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ClockAdjust(), ClockAdjust_r()

Adjust the time of a clock

Synopsis:

```
#include <sys/neutrino.h>

int ClockAdjust( clockid_t id,
                 const struct _clockadjust * new,
                 struct _clockadjust * old );

int ClockAdjust_r( clockid_t id,
                   const struct _clockadjust * new,
                   struct _clockadjust * old );
```

Arguments:

id

The ID of the clock you want to adjust. This must be `CLOCK_REALTIME`; this clock maintains the system time.

new

NULL or a pointer to a `_clockadjust` structure that specifies how to adjust the clock. Any previous adjustment is replaced.

The `_clockadjust` structure contains at least the following members:

- `long tick_nsec_inc` — the adjustment to be made on each clock tick, in nanoseconds.
- `unsigned long tick_count` — the number of clock ticks over which to apply the adjustment.

old

If not NULL, a pointer to a `_clockadjust` structure where the function can store the current adjustment (before being changed by a non-NULL *new*).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls let you gradually adjust the time of the clock specified by *id*. You can use these functions to speed up or slow down the system clock to synchronize it with another time source – without causing major discontinuities in the time flow.

The *ClockAdjust()* and *ClockAdjust_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 323) section for details.

The total time adjustment, in nanoseconds, is:

```
(new->tick_count * new->tick_nsec_inc)
```

If the current clock is ahead of the desired time, you can specify a negative *tick_nsec_inc* to slow down the clock. This is preferable to setting the time backwards with the *ClockTime()* (p. 333) kernel call, since some programs may malfunction if time goes backwards.

Picking small values for *tick_nsec_inc* and large values for *tick_count* adjusts the time slowly, while the opposite approach adjusts it rapidly. As a rule of thumb, don't try to set a *tick_nsec_inc* that exceeds the basic clock tick as set by the *ClockPeriod()* (p. 329) kernel call. This would change the clock rate by more than 100% and if the adjustment is negative, it could make the clock go backwards.

You can cancel any adjustment in progress by setting *tick_count* and *tick_nsec_inc* to 0.

In order to adjust the clock, your process must have the PROC_MGR_AID_CLOCKSET ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Blocking states:

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ClockAdjust()

If an error occurs, the function returns -1 and sets *errno* (p. 539). Any other value returned indicates success.

ClockAdjust_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, the function may return any value in the Errors section.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The clock ID isn't valid.

EPERM

The process tried to adjust the time without having the required permission;
see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ClockCycles()

Get the number of clock cycles

Synopsis:

```
#include <sys/neutrino.h>
#include <inttypes.h>

uint64_t ClockCycles( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ClockCycles()* kernel call returns the current value of a free-running 64-bit cycle counter. This is implemented on each processor as a high-performance mechanism for timing short intervals.

Several CPU architectures have an instruction that reads such a free-running counter (e.g., x86 has the RDTSC instruction). For processors that don't implement such an instruction in hardware, the kernel emulates one. This provides a lower time resolution than if the instruction is provided (e.g., 838.095345 nanoseconds on an IBM PC-compatible system).

In all cases, the [SYSPAGE_ENTRY\(qtime\)->cycles_per_sec](#) (p. 3326) field gives the number of *ClockCycles()* increments in one second.



The cycle counter wraps every number of seconds as calculated by:

```
(~(uint64_t)0) / SYSPAGE_ENTRY(qtime)->cycles_per_sec
```

Symmetric MultiProcessing systems

This function, depending on the CPU architecture, returns a value from a register that's unique to each CPU in an SMP system — for instance, the TSC (Time Stamp Counter) on an x86. These registers aren't synchronized between the CPUs. So if you call *ClockCycles()*, and then the thread migrates to another CPU and you call *ClockCycles()* again, you can't subtract the two values to get a meaningful time duration.

If you wish to use *ClockCycles()* on an SMP machine, you must use the following call to “lock” the thread to a single CPU:

```
ThreadCtl( _NTO_TCTL_RUNMASK, ... )
```

For more information, see the entry for [ThreadCtl](#) (p. 3412).

Blocking states:

This call doesn't block.

Examples:

See [SYSPAGE_ENTRY\(\)](#) (p. 3326).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

ClockId(), ClockId_r()

Get the CPU-time clock ID for a given process and thread

Synopsis:

```
#include <sys/neutrino.h>
#include <inttypes.h>

int ClockId( pid_t pid,
             int tid );

int ClockId_r( pid_t pid,
               int tid );
```

Arguments:

pid

The ID of the process that you want to get the clock ID for. If this argument is zero, the ID of the process making the call is assumed.

tid

The ID of the thread that you want to get the clock ID for, or 0 to get clock ID for the process as a whole.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ClockId()* and *ClockId_r()* kernel calls retrieve an integer CPU-time clock ID that you can pass as a `clockid_t` to *ClockTime()* (p. 333). When you pass this clock ID to *ClockTime()*, that function returns (in the location pointed to by *old*) the number of nanoseconds that the specified thread of the specified process has executed.

The *ClockId()* and *ClockId_r()* functions are identical except in the way they indicate errors. See the *Returns* (p. 328) section for details.



Instead of using these kernel calls directly, consider calling *clock_getcpuclockid()* (p. 307) or *pthread_getcpuclockid()* (p. 2295).

If the *tid* is zero, *ClockTime()* gives you the number of nanoseconds that the process as a whole has executed. On an SMP box, this number may exceed the realtime number of nanoseconds that have elapsed because multiple threads in the process can run on several CPUs at the same time.

Blocking states:

This call doesn't block.

Returns:

ClockId()

An integer that can be passed to *ClockTime()*. If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

ClockId_r()

An integer that can be passed to *ClockTime()*. This function does **NOT** set *errno*. If an error occurs, the function returns the negative of a value from the Errors section.

Errors:

ESRCH

The *pid* and/or *tid* don't exist.

Examples:

Here's how you can determine how busy a system is:

```
id = ClockId(1, 1);
for( ;; ) {
    ClockTime(id, NULL, &start);
    sleep(1);
    ClockTime(id, NULL, &stop);
    printf("load = %f%%\n", (1000000000.0 - (stop-start)) / 10000000.0);
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ClockPeriod(), ClockPeriod_r()

Get or set a clock period

Synopsis:

```
#include <sys/neutrino.h>

int ClockPeriod( clockid_t id,
                 const struct _clockperiod * new,
                 struct _clockperiod * old,
                 int reserved );

int ClockPeriod_r( clockid_t id,
                  const struct _clockperiod * new,
                  struct _clockperiod * old,
                  int reserved );
```

Arguments:

id

The clock ID of the clock; one of the following:

- **CLOCK_REALTIME** — the clock that maintains the system time.
- **CLOCK_HARMONIC** — the clock that maintains the harmonic timer boundary. Timers whose remaining time before expiry is greater than this boundary are considered for timer harmonization; for more information, see “Clocks, timers, and power management” in the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

new

NULL, or a pointer to a `_clockperiod` structure that contains the period to set the clock to. This structure contains at least the following members:

- `unsigned long nsec`—the period of the clock, in nanoseconds.



For `CLOCK_HARMONIC`, this period is in seconds, not nanoseconds.

- `long fract`—reserved for future fractional nanoseconds. Set this member to zero.

old

NULL, or a pointer to a `_clockperiod` structure where the function can store the current period (before being changed by a non-NULL *new*).

reserved

Set this argument to 0.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use the `ClockPeriod()` and `ClockPeriod_r()` kernel calls to get or set the clock period of the clock. These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 331) section for details.

-
- If you want to get the clock period, consider calling [clock_getres\(\)](#) (p. 309) instead of using these kernel calls directly.

- In order to set the clock period, your process must have the `PROCmgr_AID_CLOCKPERIOD` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If you're using adaptive partitioning, and you change the tick size of the system at runtime, do so before defining the adaptive partitioning scheduler's window size. That's because QNX Neutrino converts the window size from milliseconds to clock ticks for internal use. For more information, see the Adaptive Partitioning *User's Guide*.

All the `timer_*`(*)* calls operate with an accuracy no better than the clock period. Every moment within the microkernel is referred to as a *tick*. A tick's initial length is determined by the clock rate of your processor:

CPU clock speed:	Default value:
40MHz	1 millisecond
< 40MHz	10 milliseconds

Since a very small tick size imposes an interrupt load on the system, and can consume all available processor cycles, the kernel call limits how small a period can be specified. The lowest clock period that can currently be set on any machine is 10 microseconds.

If an attempt is made to set a value that the kernel believes to be unsafe, the call fails with an **EINVAL**. The timeslice rate (for “round-robin” and “other” scheduling policies) is always four times the clock period (this isn't changeable).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ClockPeriod()

If an error occurs, this function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

ClockPeriod_r()

EOK is returned on success. This function does **NOT** set [errno](#). If an error occurs, the function can return any value in the Errors section.

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

Invalid clock ID. A period was set which wasn't in a range considered safe.

EPERM

The process tried to change the period of the clock without having the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ClockTime(), ClockTime_r()

Get or set a clock

Synopsis:

```
#include <sys/neutrino.h>

int ClockTime( clockid_t id,
               const uint64_t * new,
               uint64_t * old );

int ClockTime_r( clockid_t id,
                 const uint64_t * new,
                 uint64_t * old );
```

Arguments:

id

The clock ID. This must be `CLOCK_REALTIME` or `CLOCK_MONOTONIC`, which is the ID of the clock that maintains the system time, or a CPU-time clock ID returned by [ClockId\(\)](#) (p. 327).

new

NULL, or a pointer to the absolute time, in nanoseconds, to set the clock to. This is used only if *id* is `CLOCK_REALTIME`.

old

NULL, or a pointer to a location where the function can store the current time (before being changed by a non-NULL *new*).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use these kernel calls to:

- get or set the system clock if *id* is `CLOCK_REALTIME`
- get the system clock if *id* is `CLOCK_MONOTONIC`

- get the CPU-time clock for a process or a particular thread in a process, if the ID is one that you obtained by calling [ClockId\(\)](#) (p. 327). On an SMP box, the time for a process may exceed the realtime number of nanoseconds that have elapsed because multiple threads in the process can run on several CPUs at the same time.

The *ClockTime()* and *ClockTime_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 334) section for details.



Instead of using these kernel calls directly, consider calling [clock_gettime\(\)](#) (p. 312) or [clock_settime\(\)](#) (p. 319).

If *new* isn't NULL, then it contains the absolute time, in nanoseconds, to set the system clock to. This affects the software clock maintained by the system. It doesn't change any underlying hardware clock that maintains the time when the system's power is turned off.



In order to set the clock, your process must have the PROCGR_AID_CLOCKSET ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141). You set the time only when the *id* is CLOCK_REALTIME.

If you call *ClockTime()* to set the time of day, the kernel checks to see if the SYS_PAGE_ENTRY(qtime) -> boot_time field is zero. If it is, the kernel sets it to the appropriate value. There's a -T option for all startup programs that prevents the setting of this field, so that the kernel will set it the first time you call *ClockTime()* to change the time of day. This is useful if the RTC hardware isn't in UTC.

Once set, the system time increments by some number of nanoseconds, based on the resolution of the system clock. You can query or change this resolution by using the [ClockPeriod\(\)](#) (p. 329) kernel call.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ClockTime()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

ClockTime_r()

EOK is returned on success. This function does **NOT** set [errno](#). If an error occurs, the function returns a value in the Errors section.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The clock ID isn't valid, or you tried to set the time for an ID other than CLOCK_REALTIME.

EPERM

The process tried to change the time without having the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process associated with this request doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

close()

Close a file

Synopsis:

```
#include <unistd.h>

int close( int filedes );
```

Arguments:

filedes

The file descriptor of the file you want to close. This can be a file descriptor returned by a successful call to [accept\(\)](#) (p. 58), [creat\(\)](#) (p. 381), [dup\(\)](#) (p. 483), [dup2\(\)](#) (p. 485), [fcntl\(\)](#) (p. 623), [modem_open\(\)](#) (p. 1672), [open\(\)](#) (p. 1905), [shm_open\(\)](#) (p. 2935), [socket\(\)](#) (p. 3075) or [sopen\(\)](#) (p. 3083).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `close()` function closes the file specified by the given file descriptor.

Returns:

Zero for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor *filedes*.

EINTR

The `close()` call was interrupted by a signal.

EIO

An I/O error occurred while updating the directory information.

ENOSPC

A previous buffered write call has failed.

ENOSYS

The *close()* function isn't implemented for the filesystem specified by *filedes*.

Examples:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes;

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        /* process file */
        close( filedes );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

closedir()

Close a directory

Synopsis:

```
#include <dirent.h>

int closedir( DIR * dirp );
```

Arguments:

dirp

A directory pointer for the directory you want to close.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *closedir()* function closes the directory specified by *dirp*, and frees the memory allocated by [opendir\(\)](#) (p. 1913).



The result of using a directory stream after calling one of the [exec*\(\)](#) (p. 550) or [spawn*\(\)](#) (p. 3089) family of functions is undefined. After a call to the [fork\(\)](#) (p. 685) function, either the parent or the child (but not both) may continue processing the directory stream using the [readdir\(\)](#) (p. 2514) and [rewinddir\(\)](#) (p. 2648) functions. If both the parent and child processes use these functions, the result is undefined. Either or both processes may call the *closedir()* function.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The *dirp* argument doesn't refer to an open directory stream.

EINTR

The *closedir()* call was interrupted by a signal.

Examples:

Get a list of files contained in the directory /home/kenny:

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

int main( void )
{
    DIR *dirp;
    struct dirent *direntp;

    dirp = opendir( "/home/kenny" );
    if( dirp != NULL ) {
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) {
                break;
            }

            printf( "%s\n", direntp->d_name );
        }

        closedir( dirp );
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

closelog()

Close the system log

Synopsis:

```
#include <syslog.h>

void closelog( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *closelog()* function closes the connection to the system log.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

_cmdfd()

Return a file descriptor for the executable file

Synopsis:

```
#include <process.h>

int _cmdfd( void );
```

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

This function returns a file descriptor for the executable file.

Returns:

A file descriptor for the executable file.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_cmdname()

Find the path used to invoke the current process

Synopsis:

```
#include <process.h>

char * _cmdname( char * buff );
```

Arguments:

buff

NULL, or a pointer to a buffer in which the function can store the path. To determine the size required for the buffer, call [fpathconf\(\)](#) (p. 699) or [pathconf\(\)](#) (p. 1936) with an argument of `_PC_PATH_MAX`, then add 1 for the terminating null character.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_cmdname()` function determines the full path that the current process was invoked from. If *buff* isn't NULL, `_cmdname()` copies the path into the buffer that *buff* points to.

Returns:

A pointer to the pathname used to load the process, or NULL if an error occurred.



Don't change the string that the returned value points to if you passed NULL for the *buff* parameter.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#include <process.h>

int main( void )
{
```

```
size_t maximum_path;
char *buff;

maximum_path = (size_t) pathconf( "/", _PC_PATH_MAX );
buff = (char* )malloc( maximum_path );

if( _cmdname( buff ) ) {
    printf( "I'm \"%s\".\n", buff );
} else {
    perror( "_cmdname() failed" );
    free (buff);
    return EXIT_FAILURE;
}

free (buff);
return EXIT_SUCCESS;
}
```

If this code is compiled into an executable named foo:

```
# ls -F /home/xyzzz/bin/foo
foo*
# /home/xyzzz/bin/foo
I'm "/home/xyzzz/bin/foo".
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

confstr()

Get configuration-defined string values

Synopsis:

```
#include <unistd.h>

size_t confstr( int name,
                char * buf,
                size_t len );
```

Arguments:

name

The system variable to query; see below.

buf

A pointer to a buffer in which the function can store the value of the system variable.

len

The length of the buffer, in bytes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *confstr()* function lets applications get or set configuration-defined string values. This is similar to the [sysconf\(\)](#) (p. 3308) function, but you use it to get string values, rather than numeric values. By default, the function queries and returns values in the system.



In order to set a configuration string, your process must have the `PROCmgr_AID_CONFSET` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The *name* argument represents the system variable to query. The values are defined in `<confname.h>`; at least the following *name* values are valid:

_CS_ARCHITECTURE

The name of the instruction set architecture for this node's CPU(s).

_CS_DOMAIN

The domain name.

_CS_HOSTNAME

The name of this node in the network.



A hostname can consist only of letters, numbers, and hyphens, and must not start or end with a hyphen. For more information, see *RFC 952*.

_CS_HW_PROVIDER

The name of the hardware manufacturer.

_CS_HW_SERIAL

Serial number associated with the hardware.

_CS_LIBPATH

A value similar to the **LD_LIBRARY_PATH** environment variable that finds all standard libraries.

_CS_LOCALE

The name of the current locale.

_CS_MACHINE

This node's hardware type.

_CS_PATH

A value similar to the **PATH** environment variable that finds all standard utilities.

_CS_RELEASE

The current OS release level.

_CS_RESOLVE

The contents of the `resolv.conf` file, excluding the domain name.

_CS_SRPC_DOMAIN

The secure RPC domain.

_CS_SYSNAME

The operating system name.

_CS_TIMEZONE

Time zone string (**TZ** style)

_CS_VERSION

The current OS version number.

The configuration-defined value is returned in the buffer pointed to by *buf*, and will be *len* bytes long, including the terminating NULL. If the value, including the terminating NULL, is greater than *len* bytes long, it's truncated to *len* - 1 bytes and terminated with a NULL character.

To find out the length of a configuration-defined value, call *confstr()* with *buf* set to NULL and *len* set to 0.

To set a configuration value:

1. OR the name of the value (e.g. `_CS_HOSTNAME`) with `_CS_SET`.
2. Put the new value in a NULL-terminated string.
3. Set the value of *len* to 0.

Returns:

A nonzero value (if a “get” is done, the value is the length of the configuration-defined value), or 0 if an error occurs (*errno* (p. 539) is set).

You can compare the *confstr()* return value against *len* to see if the configuration-defined value was truncated when retrieving a value (you can't do this when setting a value).

Errors:**EINVAL**

The *name* argument isn't a valid configuration-defined value.

EPERM

The calling process doesn't have the required permission; see [*procmgr_ability\(\)*](#) (p. 2141).

Examples:

Print information similar to that returned by the [*uname\(\)*](#) (p. 3571) function:

```
#include <unistd.h>
#include <stdio.h>
#include <limits.h>
```

```

#define BUFF_SIZE (256 + 1)

int main( void )
{
    char buff[BUFF_SIZE];

    if( confstr( _CS_SYSNAME, buff, BUFF_SIZE ) > 0 ) {
        printf( "System name: %s\n", buff );
    }

    if( confstr( _CS_HOSTNAME, buff, BUFF_SIZE ) > 0 ) {
        printf( "Host name: %s\n", buff );
    }

    if( confstr( _CS_RELEASE, buff, BUFF_SIZE ) > 0 ) {
        printf( "Release: %s\n", buff );
    }

    if( confstr( _CS_VERSION, buff, BUFF_SIZE ) > 0 ) {
        printf( "Version: %s\n", buff );
    }

    if( confstr( _CS_MACHINE, buff, BUFF_SIZE ) > 0 ) {
        printf( "Machine: %s\n", buff );
    }

    if( confstr( _CS_SET | _CS_HOSTNAME, "myhostname", 0 ) != 0 ) {
        printf( "Hostname set to: %s\n", "myhostname" );
    }

    return 0;
}

```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The *confstr()* function is part of a draft standard; its interface and/or behavior may change in the future.

connect()

Initiate a connection on a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect( int s,
             const struct sockaddr * name,
             socklen_t namelen );
```

Arguments:

s

The descriptor of the socket on which to initiate the connection.

name

The name of the socket to connect to for a **SOCK_STREAM** connection.

namelen

The length of the *name*, in bytes.

Library:

libsocket

Use the **-l socket** option to **qcc** to link against this library.

Description:

The *connect()* function establishes the connection according to the socket type specified by *s*:

SOCK_DGRAM

Specifies the peer that the socket is to be associated with. This address is the one that datagrams are to be sent to, and the only one that datagrams are to be received from.

SOCK_STREAM

This call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of that socket. Each communications space interprets *name* in its own way.

Stream sockets may successfully connect only once, whereas datagram sockets may use *connect()* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:**EADDRINUSE**

The address is already in use.

EADDRNOTAVAIL

The specified address isn't available on this machine.

EINVAL

Addresses in the specified address family cannot be used with this socket.

EALREADY

The socket is nonblocking; a previous connection attempt hasn't yet been completed.

EBADF

Invalid descriptor s.

ECONNABORTED

The *connect()* was terminated under software control.

ECONNREFUSED

The attempt to connect was forcefully rejected.

EFAULT

The *name* parameter specifies an area outside the process address space.

EHOSTUNREACH

No route to host; the host system can't be reached.

EINPROGRESS

The socket is nonblocking; the connection can't be completed immediately. It's possible to do a *select()* for completion by selecting the socket for writing.

EISCONN

The socket is already connected.

ENETUNREACH

The network isn't reachable from this host.

ETIMEDOUT

The attempt to establish a connection timed out; no connection was made.



Protocols such as TCP do not allow further connection requests on a socket after an **ECONNREFUSED** error. In such a situation, the socket must be closed and a new one created before a subsequent attempt for connection is made.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

ConnectAttach(), ConnectAttach_r()

Establish a connection between a process and a channel

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>

int ConnectAttach( uint32_t nd,
                  pid_t pid,
                  int chid,
                  unsigned index,
                  int flags );

int ConnectAttach_r( uint32_t nd,
                   pid_t pid,
                   int chid,
                   unsigned index,
                   int flags );
```

Arguments:

nd

The node descriptor of the node (e.g. ND_LOCAL_NODE for the local node) on which the process that owns the channel is running; see “[Node descriptors](#) (p. 353),” below.

pid

The process ID of the owner of the channel. If *pid* is zero, the calling process is assumed.

chid

The channel ID, returned by [ChannelCreate\(\)](#) (p. 278), of the channel to connect to the process.

index

The lowest acceptable connection ID.



Treating a connection as a file descriptor can lead to unexpected behavior. Therefore, you should OR _NTO_SIDE_CHANNEL into *index* when you create a connection. If you do this, the connection ID is returned from a different space than file descriptors; the ID is greater than any valid file descriptor.

Once created there's no difference in the use of the messaging primitives on this ID. The C library creates connections at various times without `_NTO_SIDE_CHANNEL` (e.g. during `open()`), however, it's unlikely that any applications would want to call it this way.

flags

If *flags* contains `_NTO_COF_CLOEXEC`, the connection is closed when your process calls an `exec*()` function to start a new process.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ConnectAttach()` and `ConnectAttach_r()` kernel calls establish a connection between the calling process and the channel specified by *chid* owned by the process specified by *pid* on the node specified by *nd*. Any function that passes a node descriptor can use either the value 0 or the constant `ND_LOCAL_NODE` to refer to the local node.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 353) section for details.



If a process wants other processes to communicate with it, it typically uses [name_attach\(\)](#) (p. 1842) to create a channel and associate a name with it, and the sender process uses [name_open\(\)](#) (p. 1852) to locate that name and create a connection to it.

The return value is a connection ID, which is a small `int` representing the connection. The system returns the first available connection ID starting at the value specified by the *index* argument. Any thread in the calling process can use either [MsgSendv\(\)](#) (p. 1801) to send messages or [MsgSendPulse\(\)](#) (p. 1789) to send pulses over the connection. The connection ID is used directly as a POSIX file descriptor (*fd*) when communicating with I/O Resource managers such as a filesystem manager.

If you don't OR `_NTO_SIDE_CHANNEL` into *index*, this behavior might result:

- If file descriptor 0 is in use, file descriptor 1 isn't in use, and you call `ConnectAttach()` with 0 specified for *index*, a connection ID of 1 is returned.

File descriptor 1 (i.e. connection ID 1) is used as *stdout*, which is what `printf()` writes to. If your process makes any calls to `printf()`, NULL-terminated character

strings are sent to the channel that you've connected to. Similar situations can happen with connection IDs 0 (*stdin*) and 2 (*stderr*).

- Depending on how a child process is created, it may inherit the parent's file descriptors.

Since connections are treated like file descriptors, a connection created by the parent without `_NTO_SIDE_CHANNEL` in *index* and without `_NTO_COF_CLOEXEC` in *flags*, causes a child process to inherit that connection during process creation. This inheritance is done during process creation by duplicating file descriptors.

During duplication, an `_IO_DUP` message (with `0x115` as the first 2 bytes) is sent to the receiver on the other side of the connection. The receiver won't be expecting this message.

If *index* has `_NTO_SIDE_CHANNEL` set, the *index* is ignored and the connection ID returned is the first available index in the `_NTO_SIDE_CHANNEL` space.

If a process creates multiple connections to the same channel, the system maintains a link count and shares internal kernel object resources for efficiency.

Connections are owned by the process and may be used simultaneously by any thread in the process. You can detach a connection by calling [ConnectDetach\(\)](#) (p. 367). If any threads are blocked on the channel (via *MsgSendv()*) at the time the connection is detached, the send fails and returns with an error.



Connections and connection IDs persist until you call *ConnectDetach()*, even if the other process dies.

The connection is strictly local (i.e. it doesn't resolve across the network) and is resolved on the first use of the connection ID.

Blocking states

These calls don't block.

Node descriptors

The *nd* (node descriptor) is a temporary numeric description of a remote node. For more information, see the Qnet Networking chapter of the *System Architecture* guide.

To:	Use this function:
Compare two <i>nd</i> objects	ND_NODE_CMP() (p. 1874)
Convert a <i>nd</i> to text	netmgr_ndtostr() (p. 1877)
Convert text to a <i>nd</i>	netmgr_strtond() (p. 1883)

Returns:

The only difference between these functions is the way they indicate errors:

ConnectAttach()

A connection ID that's used by the message primitives. If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

ConnectAttach_r()

A connection ID that's used by the message primitives. This function does **NOT** set *errno*. If an error occurs, the function returns the negative of a value from the Errors section.

Errors:**EAGAIN**

Insufficient resources.

EINVAL

One of the following:

- The value of *index* is outside the range of valid connection IDs. This error applies only to file descriptors, not side channels (there's no limit on the range for side channels).
- The combination of bits in the *flags* argument is invalid.

EMFILE

The process has no unused file descriptors, or no file descriptors greater than or equal to *index* are available.

ENOREMOTE

The node indicated by *nd* doesn't exist.

ENXIO

The process indicated by *pid* is no longer valid. It's going away, has core dumped, or become a zombie.

EPERM

The process that created the channel used the `_NTO_CHF_PRIVATE` flag to mark it as being private. For more information, see [ChannelCreate\(\)](#) (p. 278).

ESRCH

The process indicated by *pid*, or the channel indicated by *chid*, doesn't exist.

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ConnectClientInfo(), ConnectClientInfo_r()

Store information about a client connection

Synopsis:

```
#include <sys/neutrino.h>

int ConnectClientInfo( int scoid,
                      struct _client_info * info
                      int ngroups );

int ConnectClientInfo_r( int scoid,
                        struct _client_info * info
                        int ngroups );
```

Arguments:

scoid

A server connection ID that identifies the client process that you want to get information about. This client is typically a process that's made a connection to the server to try to access a resource. You can get it from the [_msg_info](#) (p. 1731) argument to [MsgReceive\(\)](#) (p. 1771) or [MsgInfo\(\)](#) (p. 1745).

info

A pointer to a [_client_info](#) (p. 357) structure that the function can fill with information about the client. For more information, see below.

ngroups

The size of the caller's *group*list in the credential part of the [_client_info](#) structure. If you make it smaller than NGROUPS_MAX, you might get information only about a subset of the groups.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

These calls get information about a client connection identified by *scoid*, and store it in the buffer that *info* points to.



These functions can handle no more than NGROUPS_MAX supplementary groups. For more groups, use [ConnectClientInfoExt\(\)](#) (p. 363) instead.

The *ConnectClientInfo()* and *ConnectClientInfo_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 358) section for details.

A server uses these functions to determine whether or not a client has permission to access a resource. For example, in a resource manager, it would be called on an [open\(\)](#) (p. 1905) connection request.

_client_info structure

The `_client_info` structure has at least the following members:

uint32_t nd

The client's node descriptor, a temporary numeric description of a remote node; ND_LOCAL_NODE (or 0) is the descriptor for the local node. For more information, see the Qnet Networking chapter of the *System Architecture* guide.

To:	Use this function:
Compare two <i>nd</i> objects	ND_NODE_CMP() (p. 1874)
Convert a <i>nd</i> to text	netmgr_ndtostr() (p. 1877)
Convert text to a <i>nd</i>	netmgr_strtond() (p. 1883)

pid_t pid

The client's process ID.

pid_t sid

Used internally by Qnet.

flags

- `_NTO_CI_ENDIAN_BIG` — the client is on a big-endian machine.
- `_NTO_CI_BKGND_PGRP` — the client's process group is in the background.
- `_NTO_CI_ORPHAN_PGRP` — the client's process group has been orphaned.
- `_NTO_CI_STOPPED` — the client is stopped.
- `_NTO_CI_UNABLE` — the client doesn't have the required abilities; see [ConnectClientInfoAble\(\)](#) (p. 360), [procmgr_ability\(\)](#) (p. 2141), and [procmgr_ability_create\(\)](#) (p. 2152).

- `_NTO_CI_FULL_GROUPS` — indicates that the client information contains the full group list. Set by *ConnectClientInfoExt()*, but not *ConnectClientInfo()*.

struct `_cred_info cred`

The user and group ID credentials; a `_cred_info` structure that includes at least the following members:

`uid_t ruid`

The real user ID of the sending process.

`uid_t euid`

The effective user ID of the sending process.

`uid_t suid`

The saved user ID of the sending process.

`gid_t rgid`

The real group ID of the sending process.

`gid_t egid`

The effective group ID of the sending process.

`gid_t sgid`

The saved group ID of the sending process.

`uint32_t ngroups`

The number of groups actually stored in *grouplist*.

`gid_t grouplist[NGROUPS_MAX]`

The supplementary group IDs of the sending process.

The *ngroups* argument to *ConnectClientInfo()* indicates the size of the *grouplist* array. If the group array size is zero, the *ngroups* member of the `_cred_info` is set to the number of groups available.

Returns:

The only difference between these functions is the way they indicate errors:

ConnectClientInfo()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

ConnectClientInfo_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, the function can return any value in the Errors section.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

Process doesn't have a connection *scoId*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ConnectClientInfoAble()

Store extended information about a client connection, including abilities

Synopsis:

```
#include <sys/neutrino.h>

struct _client_able {
    unsigned ability;
    unsigned flags;
    uint64_t range_lo;
    uint64_t range_hi;
};

int ConnectClientInfoAble( int scoId,
                          struct _client_info **info_pp,
                          const int flags,
                          struct _client_able abilities[],
                          const int nable );
```

Arguments:

scoId

A server connection ID that identifies the client process that you want to get information about. This client is typically a process that's made a connection to the server to try to access a resource. You can get it from the [_msg_info](#) (p. 1731) argument to [MsgReceive\(\)](#) (p. 1771) or [MsgInfo\(\)](#) (p. 1745).

info

The address of a location where the function can store a pointer to a `_client_info` structure that the function can fill with information about the client. For more information, see [ConnectClientInfoExt\(\)](#) (p. 363).

flags

Zero or more of the following:

- `_NTO_CLIENTINFO_GETGROUPS` — get the supplementary group IDs

able

An array of [struct _client_able](#) (p. 361) structures that specifies the abilities that the client must have; see below.

nable

The number of entries in the *able* array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This call gets information about a client connection identified by *scoId*, allocates a buffer, stores the information in the buffer, and stores a pointer to the buffer in the location that *info* points to. Use [ClientInfoExtFree\(\)](#) (p. 304) to free this buffer.

A server uses this function to determine whether or not a client has the required abilities; see [procmgr_ability\(\)](#) (p. 2141), and [procmgr_ability_create\(\)](#) (p. 2152).

The *ConnectClientInfoAble()* function is a specialized version of [ConnectClientInfo\(\)](#) (p. 356).

_client_able structure

The `_client_able` structure has at least the following members:

ability

The numeric identifier for the ability; either a `PROCMGR_AID_*` value (see [procmgr_ability\(\)](#) (p. 2141)), or an ability that the server created by calling [procmgr_ability_create\(\)](#) (p. 2152).

flags

Flags that specify the domain of the ability; a bitwise OR of zero or more of the following:

- `PROCMGR_ADN_ROOT` — the ability is for privileged processes.
- `PROCMGR_ADN_NONROOT` — the ability is for unprivileged processes.

range_lo

The lower limit on the range of the ability.

range_hi

The upper limit on the range of the ability.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The process doesn't have a connection *soid*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ConnectClientInfoExt()

Store extended information about a client connection

Synopsis:

```
#include <sys/neutrino.h>

int ConnectClientInfoExt( int scoid,
                        struct _client_info ** info,
                        int flags );
```

Arguments:

scoid

A server connection ID that identifies the client process that you want to get information about. This client is typically a process that's made a connection to the server to try to access a resource. You can get it from the [_msg_info](#) (p. 1731) argument to [MsgReceive\(\)](#) (p. 1771) or [MsgInfo\(\)](#) (p. 1745).

info

The address of a location where the function can store a pointer to a [_client_info](#) (p. 364) structure that the function can fill with information about the client. For more information, see below.

flags

Zero or more of the following:

- [_NTO_CLIENTINFO_GETGROUPS](#) — get the supplementary group IDs

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This call gets information about a client connection identified by *scoid*, allocates a buffer, stores the information in the buffer, and stores a pointer to the buffer in the location that *info* points to. Use [ClientInfoExtFree\(\)](#) (p. 304) to free this buffer.

A server uses this function to determine whether or not a client has permission to access a resource. For example, in a resource manager, it would be called on an [open\(\)](#) (p. 1905) connection request.

The *ConnectClientInfoExt()* function is an extended version of [ConnectClientInfo\(\)](#) (p. 356) that can handle more than NGROUPS_MAX supplementary groups.

_client_info structure

The *_client_info* structure has at least the following members:

uint32_t nd

The client's node descriptor, a temporary numeric description of a remote node; ND_LOCAL_NODE (or 0) is the descriptor for the local node. For more information, see the Qnet Networking chapter of the *System Architecture* guide.

To:	Use this function:
Compare two <i>nd</i> objects	ND_NODE_CMP() (p. 1874)
Convert a <i>nd</i> to text	netmgr_ndtostr() (p. 1877)
Convert text to a <i>nd</i>	netmgr_strtond() (p. 1883)

pid_t pid

The client's process ID.

pid_t sid

Used internally by Qnet.

flags

- **_NTO_CI_ENDIAN_BIG** — the client is on a big-endian machine.
- **_NTO_CI_BKGND_PGRP** — the client's process group is in the background.
- **_NTO_CI_ORPHAN_PGRP** — the client's process group has been orphaned.
- **_NTO_CI_STOPPED** — the client is stopped.
- **_NTO_CI_UNABLE** — the client doesn't have the required abilities; see [ConnectClientInfoAble\(\)](#) (p. 360), [procmgr_ability\(\)](#) (p. 2141), and [procmgr_ability_create\(\)](#) (p. 2152).
- **_NTO_CI_FULL_GROUPS** — indicates that the client information contains the full group list. Set by *ConnectClientInfoExt()*, but not *ConnectClientInfo()*.

struct _cred_info cred

The user and group ID credentials; a `_cred_info` structure that includes at least the following members:

uid_t ruid

The real user ID of the sending process.

uid_t euid

The effective user ID of the sending process.

uid_t suid

The saved user ID of the sending process.

gid_t rgid

The real group ID of the sending process.

gid_t egid

The effective group ID of the sending process.

gid_t sgid

The saved group ID of the sending process.

uint32_t ngroups

The number of groups actually stored in *grouplist*.

gid_t grouplist[]

The supplementary group IDs of the sending process.



This list doesn't include the primary group ID (unless it was also added as a supplementary group).

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The process doesn't have a connection *soid*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ConnectDetach(), ConnectDetach_r()

Break a connection between a process and a channel

Synopsis:

```
#include <sys/neutrino.h>

int ConnectDetach( int coid );
int ConnectDetach_r( int coid );
```

Arguments:

coid

The connection ID of the connection you want to break.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ConnectDetach()* and *ConnectDetach_r()* kernel calls detach the connection specified by the *coid* argument.

When the connection is detached:

- If a client thread is SEND-blocked on the connection (e.g., it's called [MsgSend\(\)](#) (p. 1781), but the server hasn't received the message), the send fails and returns with an error.
- If a client thread is REPLY-blocked on the connection (e.g., it's called *MsgSend()*, and the server received the message), the kernel sends an unblock pulse to the server.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 367) section for details.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ConnectDetach()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

ConnectDetach_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, the function returns a value in the Errors section.

Errors:**EINVAL**

The connection specified by *coid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ConnectFlags(), ConnectFlags_r()

Modify the flags associated with a connection

Synopsis:

```
#include <sys/neutrino.h>

int ConnectFlags( pid_t pid,
                  int coid,
                  unsigned mask,
                  unsigned bits );

int ConnectFlags_r( pid_t pid,
                    int coid,
                    unsigned mask,
                    unsigned bits );
```

Arguments:

pid

The ID of the process that the connection ID belongs to, or 0 for the current process.

coid

The ID of the connection whose flags you want to modify.

mask

A bitmap that indicates which bits are to be modified in the flags. The bits currently defined include:

- `_NT0_COF_CLOEXEC` — close the connection if the process calls an `exec*()` function to start a new process.
- `_NT0_COF_INSECURE` — the kernel sets this flag on a connection if the client program replaces its process image (e.g., by calling [exec*\(\)](#) (p. 487)) with a `setuid` or `setgid` binary. Until the process that owns the channel clears this flag, calls to [MsgDeliverEvent\(\)](#) (p. 1736) fail with an error of `EACCES`.

bits

The new value of the flags.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ConnectFlags()* and *ConnectFlags_r()* kernel calls modify flags associated with the specified connection. These kernel calls don't block.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 370) section for details.

You need to initialize the bits that correspond to the flag in both the *mask* and *bits* arguments:

- If the bit in the *mask* is 1, and the bit in the *bits* is 1, the function turns the flag on.
- If the bit in the *mask* is 1, and the bit in the *bits* is 0, the function turns the flag off.
- If the bit in the *mask* is 0, the function doesn't change the current value of the flag.



In order to change the connection flags on a *coid* for another process with a different user ID, your process must have the `PROCmgr_AID_CONNECTION` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The only difference between these functions is the way they indicate errors:

ConnectFlags()

The previous value of the flags associated with the connection. If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

ConnectFlags_r()

The previous value of the flags associated with the connection. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EBADF**

The *coid* isn't a valid connection ID for the process.

EINVAL

The *mask* includes invalid bits.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process ID is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ConnectServerInfo(), ConnectServerInfo_r()

Get information about a server connection

Synopsis:

```
#include <sys/neutrino.h>

int ConnectServerInfo( pid_t pid,
                      int coid,
                      struct _server_info* info );

int ConnectServerInfo_r( pid_t pid,
                        int coid,
                        struct _server_info* info );
```

Arguments:

pid

The process ID of the owner of the connection.

coid

The connection ID of the connection.

info

NULL, or a pointer to a [_server_info](#) (p. 373) structure where the function can store information about the connection. For more information, see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ConnectServerInfo()* and *ConnectServerInfo_r()* kernel calls get information about the connection *coid* owned by process *pid*, and store it in the structure pointed to by *info*. If the process doesn't have a connection *coid*, the call scans for the next higher connection and returns it if present. Otherwise, -1 is returned. If you wish to check for the existence of an exact connection, you must compare the returned connection with the *coid* you requested.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 374) section for details.

If the *info* argument is NULL, *ConnectServerInfo()* ignores connections with dead servers and skips to the next coid. If *info* is non-NULL, the function fills in the *_server_info* structure; for connections with dead servers, it turns on the *_NTO_COF_DEAD* bit in the *flags* field of the structure.

***_server_info* structure**

The *_server_info* structure that *info* points to includes at least the following members:

uint32_t nd

The server's node descriptor, a temporary numeric description of a remote node; *ND_LOCAL_NODE* (or 0) is the descriptor for the local node. For more information, see the Qnet Networking chapter of the *System Architecture* guide.

To:	Use this function:
Compare two <i>nd</i> objects	<i>ND_NODE_CMP()</i> (p. 1874)
Convert a <i>nd</i> to text	<i>netmgr_ndtostr()</i> (p. 1877)
Convert text to a <i>nd</i>	<i>netmgr_strtond()</i> (p. 1883)

pid_t pid

The server's process ID.

int32_t chid

The server's channel ID.

int32_t scoid

The server's connection ID.

int32_t coid

The client's connection ID.

int16_t flags

Flags that indicate the properties of the connection. The bits include:

- *_NTO_COF_CLOEXEC* — the connection will be closed when your process calls an *exec*()* function to start a new process.
- *_NTO_COF_DEAD* — the connection is to a dead server.
- *_NTO_COF_NOSHARE* — some internal data structures aren't shared.

- `_NTO_COF_NONBLOCK` — don't block waiting if all the send headers are in use; for asynchronous messaging.
- `_NTO_COF_ASYNC` — the connection is for asynchronous messaging.
- `_NTO_COF_GLOBAL` — the connection is to a global channel.

Returns:

The only difference between these functions is the way they indicate errors:

ConnectServerInfo()

A matched *coid*. If an error occurs, the function returns -1 and sets *errno* (p. 539).

ConnectServerInfo_r()

A matched *coid*. This function does **NOT** set *errno*. If an error occurs, the function returns the negative of a value from the Errors section.

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

Process *pid* doesn't have a connection $\geq$ *coid*.

ESRCH

The process indicated by *pid* doesn't exist.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

copysign(), copysignf()

Copy the sign bit from one number to another

Synopsis:

```
#include <math.h>

double copysign ( double x,
                  double y);

float copysignf ( float x,
                  float y );
```

Arguments:

x

The number to use the magnitude of.

y

The number to use the sign of.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *copysign()* and *copysignf()* functions return the magnitude of *x* and the sign bit of *y*.

If *x* is NAN, the function produces NAN with the sign of *y*.

Returns:

The magnitude of *x* and the sign bit of *y*.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c;
```

```
a = 27.0;
b = -5;
c = copysign(a, b);
printf("The magnitude of %f and sign of %f gives %f\n",
      a, b, c);

return(0);
}
```

produces the output:

```
The magnitude of 27.000000 and sign of -5.000000 gives -27.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

cos(), cosf(), cosl()

Compute the cosine of an angle

Synopsis:

```
#include <math.h>

double cos( double x );
float cosf( float x );
long double cosl( long double x );
```

Arguments:

x

The angle, in radians, for which you want to compute the cosine.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the cosine of *x* (specified in radians).



An argument with a large magnitude may yield results with little or no significance.

Returns:

The cosine of *x*. When *x* is +/-Inf, these functions return NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main( void )
{
    double value;

    value = cos( M_PI );
    printf( "value = %f\n", value );

    return EXIT_SUCCESS;
}
```

produces the output:

```
value = -1.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

cosh(), coshf(), coshl()

Compute the hyperbolic cosine

Synopsis:

```
#include <math.h>

double cosh( double x );
float coshf( float x );
long double coshl( long double x );
```

Arguments:

x

The angle, in radians, for which you want to compute the hyperbolic cosine.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the hyperbolic cosine (specified in radians) of *x*. A range error occurs if the magnitude of *x* is too large.

Returns:

The hyperbolic cosine of *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", cosh(.5) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
1.127626
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

creat(), creat64()

Create and open a file (low-level)

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat( const char* path,
           mode_t mode );

int creat64( const char* path,
             mode_t mode );
```

Arguments:

path

The path of the file you want to open.

mode

The access permissions that you want to use. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *creat()* and *creat64()* functions create and open the file specified by *path* with the given *mode*.

Calling *creat()* is the same as:

```
open( path, O_WRONLY | O_CREAT | O_TRUNC, mode );
```

Similarly, calling *creat64()* is the same as:

```
open64( path, O_WRONLY | O_CREAT | O_TRUNC | O_LARGEFILE, mode );
```

If *path* exists and is writable, it's truncated to contain no data, and the existing *mode* setting isn't changed.

If *path* doesn't exist, it's created with the access permissions specified by the *mode* argument. The access permissions for the file or directory are specified as a combination of the bits defined in `<sys/stat.h>`.

Returns:

A file descriptor on success, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EACCES**

Indicates one of the following permission problems:

- Search permission is denied for one of the components in the *path*.
- The file specified by *path* exists, and the permissions specified by *mode* are denied.
- The file specified by *path* doesn't exist, and the file couldn't be created because write permission is denied for the parent directory.

EBADFSYS

While attempting to open *path*, the file itself or a component of its path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to or while the directory was being updated. The filesystem must be repaired before proceeding.

EBUSY

The file specified by *path* is a block special device that's already open for writing, or *path* names a file on a filesystem mounted on a block special device that is already open for writing.

EINTR

The call to *creat()* was interrupted by a signal.

EISDIR

The file specified by *path* is a directory and the file creation flags specify write-only or read/write access.

ELOOP

Too many levels of symbolic links.

EMFILE

This process is using too many file descriptors.

ENAMETOOLONG

The length of *path* exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENFILE

Too many files are currently open in the system.

ENOENT

Either the path prefix doesn't exist, or the *path* argument points to an empty string.

ENOSPC

The directory or filesystem that would contain the new file doesn't have enough space available to create a new file.

ENOSYS

The *creat()* function isn't implemented for the filesystem specified by *path*.

ENOTDIR

A component of the path prefix isn't a directory.

EROFS

The file specified by *path* resides on a read-only filesystem.

Examples:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main( void )
{
    int filedes;

    filedes = creat( "file",
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
    if( filedes != -1 ) {
        /* process file */

        close( filedes );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

creat() is [POSIX 1003.1](#) (p. 46); *creat64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

crypt()

Encrypt a password

Synopsis:

```
#include <unistd.h>

char * crypt( const char * key,
              const char * salt );
```

Arguments:

key

A NUL-terminated string (normally a password typed by a user).

salt

A two-character string chosen from the set [a-zA-Z0-9./]. This function doesn't validate the values for *salt*, and values outside this range may cause undefined behavior. This string is used to perturb the algorithm in one of 4096 different ways.

Library:

liblogin

Use the `-l login` option to `qcc` to link against this library.



We've deprecated the version of this function in **libc**. You should link against **liblogin** if your application authenticates users against the `/etc/passwd` and `/etc/shadow` files.

Description:

The *crypt()* function performs password encryption. It's based on the Data Encryption Standard algorithm, and also includes code to deter key search attempts.



This function checks only the first eight characters of *key*.

You can obtain a 56-bit key by taking the lowest 7 bits of *key*. The 56-bit key is used to repeatedly encrypt a constant string (usually all zeroes).

For license information, see the Third Party License Terms List at <http://licensing.qnx.com/third-party-terms/>.

Returns:

A pointer to the 13-character encrypted value, or `NULL` on failure. The first two characters of the encrypted value are the *salt* itself.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The return value points to static data that's overwritten by each call to *crypt()*.

ctermid()

Generate the path name of the current controlling terminal

Synopsis:

```
#include <stdio.h>

char * ctermid( char * s );
```

Arguments:

s

NULL, or a pointer to a buffer in which the function can store the path name of the controlling terminal. This string should be at least `L_ctermid` characters long (see `<stdio.h>`).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ctermid()` function generates a string that contains the path name of the current controlling terminal for the calling process.



If the argument `s` is NULL, the string is built in a static buffer, and the function returns a pointer to the buffer.

Returns:

A pointer to the path name of the controlling terminal, or a pointer to a null string if the function can't locate the controlling terminal.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "Controlling terminal is %s\n", ctermid( NULL ) );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Read the <i>Caveats</i>

Caveats:

The *ctermid()* function isn't thread-safe if the *s* argument is NULL.

ctime(), ctime_r()

Convert calendar time to local time

Synopsis:

```
#include <time.h>

char* ctime( const time_t* timer );
char* ctime_r( const time_t* timer,
               char* buf );
```

Arguments:

timer

A pointer to a `time_t` object that contains the time that you want to convert to a string.

buf

(*ctime_r()* only) A buffer in which *ctime_r()* can store the resulting string. This buffer must be large enough to hold at least 26 characters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ctime()* and *ctime_r()* functions convert the time pointed to by *timer* to local time and formats it as a string containing exactly 26 characters in the form:

Tue May 7 10:40:27 2002\n\0

This function:	Is equivalent to calling:
<i>ctime()</i>	<code>asctime(localtime (timer));</code>
<i>ctime_r()</i>	<code>asctime_r(localtime (timer), buf);</code>



The `ctime()` function places the result string in a static buffer that's reused each time you call `ctime()` or `asctime()` (p. 155). Calling `gmtime()` (p. 1005) or `localtime()` (p. 1531) could also change the date in this static buffer.

The result string for `ctime_r()` is contained in the buffer pointed to by `buf`.

All fields have a constant width. The newline character '`\n`' and NUL character '`\0`' occupy the last two positions of the string.

Whenever the `ctime()` or `ctime_r()` functions are called, the `tzset()` (p. 3547) function is also called.

The calendar time is usually obtained by using the `time()` function. That time is Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

You typically set the time on the computer with the `date` command to reflect Coordinated Universal Time (UTC), and then use the **TZ** environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Returns:

A pointer to the string containing the formatted local time, or NULL if an error occurs.

Classification:

`ctime()` is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); `ctime_r()` is [POSIX 1003.1 TSF](#) (p. 46)

Table 3: `ctime()`

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

Table 4: `ctime_r()`

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

The [asctime\(\)](#) (p. 155) and *ctime()* functions place their results in a static buffer that's reused for each call to *asctime()* or *ctime()*.

Chapter 6

D

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<code>_DEVCTL_DATA()</code>	iofunc_devctl() (p. 1257)
<code>_DEXTRA_FIRST()</code>	readdir() (p. 2514)
<code>_DEXTRA_NEXT()</code>	readdir() (p. 2514)
<code>_DEXTRA_VALID()</code>	readdir() (p. 2514)
<code>__DIOF()</code>	devctl() (p. 403)
<code>__DION()</code>	devctl() (p. 403)
<code>__DIOT()</code>	devctl() (p. 403)
<code>__DIOTF()</code>	devctl() (p. 403)

The following have been deprecated:

Instead of:	Use:
<code>drem()</code> , <code>dremf()</code>	remainder() (p. 2554)

This chapter describes the API components whose names begin with “D”.

daemon()

Run a process in the background

Synopsis:

```
#include <stdlib.h>

int daemon( int nochdir,
            int noclose );
```

Arguments:

nochdir

If this argument is 0, the current working directory is changed to the root directory (/).

noclose

If this argument is 0, standard input, standard output, and standard error are redirected to /dev/null.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *daemon()* function allows programs to detach themselves from the controlling terminal and run in the background as system daemons.

This function calls [fork\(\)](#) (p. 685) and [setsid\(\)](#) (p. 2907).

The controlling terminal behaves as in Unix System V, Release 4. An *open()* on a terminal device not already associated with another session causes the device to become the controlling terminal for that process.



The High Availability Manager can see death messages only from self-attached entities, processes that terminate abnormally, and tasks that are running in session 1. The *daemon()* function doesn't put the caller into that session; either make your process into a self-attached entity, or use [procmgr_daemon\(\)](#) (p. 2156) instead if you want to use your application with the HAM.

The HAM automatically switches to monitoring the new process that *daemon()* creates, if the original process was a self-attached entity. For more information, see the High Availability Framework *Developer's Guide*.

Returns:

Zero for success, or -1 if an error occurs (*errno* (p. 539) is set).

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

Caveats:

Currently, *daemon()* is supported only in single-threaded applications. If you create a thread and then call *daemon()*, the function returns -1 and sets *errno* to ENOSYS.

daylight

Indicator of support for daylight saving time in the locale

Synopsis:

```
#include <time.h>

unsigned int daylight;
```

Description:

This global variable has a value of 1 when daylight saving time is supported in this locale, and 0 otherwise. Whenever you call a time function, [tzset\(\)](#) (p. 3547) is called to set the variable, based on the current time zone.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

DebugBreak()

Enter the process debugger

Synopsis:

```
#include <sys/neutrino.h>

void DebugBreak( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *DebugBreak()* kernel call activates the process debugger if you're debugging the calling process. If not, it sends a SIGTRAP signal to the process.

Blocking states

None.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

If you call *DebugBreak()* from an interrupt handler, it'll activate the *kernel* debugger (if it's present in your boot image) or send the process a SIGTRAP signal.

DebugKDBreak()

Enter the kernel debugger

Synopsis:

```
#include <sys/neutrino.h>

void DebugKDBreak( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *DebugKDBreak()* kernel call activates the kernel debugger if it's present in your boot image. If not, nothing happens.

Blocking states

None.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

DebugKDOutput()

Print text with the kernel debugger

Synopsis:

```
#include <sys/neutrino.h>

void DebugKDOutput( const char* str,
                    size_t size );
```

Arguments:

str

The string that you want to print.

size

The number of characters to print.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `DebugKDBreak()` kernel call causes the kernel debugger to print *size* characters from *str* if the kernel debugger is present in your boot image. If it isn't in your boot image, nothing happens.

When, where, and how the kernel debugger displays this message depends on which host debugger you're using.

Blocking states

None.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

delay()

Suspends a calling thread for a given length of time

Synopsis:

```
#include <unistd.h>

unsigned int delay( unsigned int duration );
```

Arguments:

duration

The number of milliseconds for which to suspend the calling thread from execution.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `delay()` function suspends the calling thread for *duration* milliseconds.



The suspension time may be greater than the requested amount, due to the nature of time measurement (see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*), or due to the scheduling of other, higher-priority threads by the system.

Returns:

0 for success, or the number of unslept milliseconds if interrupted by a signal.

Errors:

If an error occurs, [`errno`](#) (p. 539) is set to:

EAGAIN

No timer resources were available to satisfy the request.

Examples:

```
#include <unistd.h>
#include <stdlib.h>

void play_sound( void )
{
    ...
}

void stop_sound( void )
{
    ...
}

int main( void )
{
    play_sound();
    delay( 500 ); /* delay for 1/2 second */
    stop_sound();

    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

devctl()*Control a device***Synopsis:**

```
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>

int devctl( int filedes,
            int dcmd,
            void * dev_data_ptr,
            size_t n_bytes,
            int * dev_info_ptr );

int devctlv( int filedes,
             int dcmd,
             int sparts,
             int rparts,
             const iov_t *sv,
             const iov_t *rv,
             int *dev_info_ptr);
```

Arguments:***filedes***

A file descriptor that you obtained by opening the device.

dcmd

A device-specific command for the process managing the open device. The set of valid device-control commands, the associated data interpretation, the returned *dev_info_ptr* values, and the effect of the command on the device all depend on the device driver.

For:	See:
General information	" Device-control commands (p. 405)," below
Commands for manipulating processes	"Controlling processes via the /proc filesystem" in the Processes chapter of the QNX Neutrino <i>Programmer's Guide</i>
Specific commands	<sys/dcmd_*.h> header files

dev_data_ptr

(*devctl()* only) Depending on the command, this argument is one of:

- a pointer to a buffer containing data to be passed to the driver
- a receiving area for data coming from the driver
- both of the above
- NULL.

n_bytes

(*devctl()* only) The size of the data to be sent to the driver, or the maximum size of the data to be received from the driver.

sparts

(*devctlv()* only) The number of elements in the *sv* array.

rparts

(*devctlv()* only) The number of elements in the *rv* array.

sv

(*devctlv()* only) An array of buffers that hold the data to send to the driver.

rv

(*devctlv()* only) An array of buffers in which the driver can store data to return to the caller.

dev_info_ptr

NULL, or a pointer to a location that the device to can use to return additional status information instead of just success or failure. The data returned via *dev_info_ptr* depends on the device driver.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *devctl()* function sends the device-specific command *dcmd* to the process managing the device opened as *filedes*. For example, you can send commands to specify properties for devices such as keyboards, sound cards or serial ports.

The *devctlv()* function is similar to *devctl()*, but uses I/O vectors for passing data to and from the driver.

Device-control commands

Use these macros to set up the device-control commands:

__DIOF(class, cmd, data)

Get information from the device.

__DION(class, cmd)

A command with no associated data.

__DIOT(class, cmd, data)

Pass information to the device.

__DIOTF(class, cmd, data)

Pass some information to the device, and get some from it.

The arguments to these macros are:

class

The major category for the command. The device-control commands are divided into the following classes to make organization easier:

- **_DCMD_ALL** — Common (all I/O servers).
- **_DCMD_CAM** — Low-level (Common Access Method) devices, such as disks or CD-ROMs.
- **_DCMD_CHR** — Character devices.
- **_DCMD_FSYS**, **_DCMD_BLK** — Filesystem/block I/O managers.
- **_DCMD_INPUT** — Input devices.
- **_DCMD_IP** — Internet Protocol.
- **_DCMD_MEM** — Memory card.
- **_DCMD_MISC** — Miscellaneous commands.
- **_DCMD_MIXER** — Mixer (Audio).
- **_DCMD_NET** — Network devices.
- **_DCMD_PROC** — Process manager.

cmd

The specific command in the class.

data

The type of data to pass to and/or from the device. The *dev_data_ptr* argument to *devctl()* must be a pointer to this type of data, and *n_bytes* is usually the size of this type of data.



The size of the structure that's passed as the last field to the `__DIO*` macros **must** be less than $2^{14} == 16$ KB. Anything larger than this interferes with the upper two directional bits.

Resource managers can use the following macros, which are defined in `<devctl.h>`, when handling commands:

get_device_command(cmd)

Extract the class and the specific device command from *cmd* (i.e. strip off the data type and the direction).

get_device_direction(cmd)

Get the direction of the command (DEVDIR_TO, DEVDIR_FROM, DEVDIR_TOFROM, or DEVDIR_NONE).

Returns:

EOK

Success.

EAGAIN

The *devctl()* command couldn't be completed because the device driver was in use by another process, or the driver was unable to carry out the request due to an outstanding command in progress.

EBADF

Invalid open file descriptor, *filesdes*.

EINTR

The *devctl()* function was interrupted by a signal.

EINVAL

The device driver detected an error in *dev_data_ptr* or *n_bytes*.

EIO

The *devctl()* function couldn't complete because of a hardware error.

ENOTTY

The device doesn't support the *dcmd* command.

ENXIO

No such device or address. For example, there's no media available to be loaded in the device represented by the file descriptor.

EPERM

The process doesn't have sufficient permission to carry out the requested command.

Examples:

Example 1: Setting RTS on a serial port

Here's a quick example of setting and unsetting RTS (Request to Send) on a serial port:

```
/* For "devctl()" */
#include <devctl.h>
#include <sys/dcmd_chr.h>

/* For "open()" */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* For Errors */
#include <stdlib.h>
#include <stdio.h>

int check_RTS(int fd);

int main(void)
{
    int data = 0, fd, error;

    if((fd = open ("/dev/ser2", O_RDONLY)) == -1)
    {
        fprintf(stderr,
            "Error with open() on /dev/ser2. Make sure it exists.\n");
        perror (NULL);
        exit(EXIT_FAILURE);
    }

    check_RTS(fd);

    /* Let's turn ON RTS now. */
    data = _CTL_RTS_CHG | _CTL_RTS;

    if (error = devctl (fd, DCMD_CHR_SERCTL, &data,
        sizeof(data), NULL))
    {
        fprintf(stderr, "Error setting RTS: %s\n",
            strerror ( error ));
        exit(EXIT_FAILURE);
    }
    /* RTS should now be ON. */

    check_RTS(fd);

    sleep (2);

    /* Now let's turn RTS OFF. */
    data = _CTL_RTS_CHG | 0;

    if (error = devctl (fd, DCMD_CHR_SERCTL, &data,
        sizeof(data), NULL))
    {
        fprintf(stderr, "Error setting RTS: %s\n",
            strerror ( error ));
        exit(EXIT_FAILURE);
    }
}
```

```

        /* RTS should now be OFF. */
        check_RTS(fd);
        close(fd);
        return (1);
    }

int check_RTS(int fd)
{
    int data = 0, error;

    /*
     * Let's see if RTS is set, tell devctl() we're requesting
     * line status information and devctl() then assigns data
     * the line status information for us. Too easy.
     */
    if (error = devctl (fd, DCMD_CHR_LINESTATUS, &data,
                      sizeof(data), NULL))
    {
        fprintf(stderr, "Error setting RTS: %s\n",
                strerror ( error ));
        exit(EXIT_FAILURE);
    }

    if (data & _LINESTATUS_SER_RTS)
        printf("RTS is SET!\n");
    else
        printf("RTS is NOT set\n");

    return(1);
}

```

The two main areas of interest are the setting of *data* and the *devctl()* call. The *data* variable is used for both sending and receiving data.

When setting RTS, *data* is assigned a value that's sent to the device via *devctl()*.

If <i>data</i> equals:	RTS is turned:
<code>_CTL_RTS_CHG _CTL_RTS</code>	ON
<code>_CTL_RTS_CHG</code>	OFF

When checking to see if RTS is set, we call *devctl()* with *dcmd* set to the `DCMD_CHR_LINESTATUS` macro and *data* containing any value (zero is clean). The *devctl()* function returns with *data* containing the Line Status value. This then can be used to determine what lines are set on that device. In our example, we check against `_LINESTATUS_SER_RTS`.

To find out what values to use with different `DCMD_*` commands, look in the appropriate `<sys/dcmd_*.h>` header file. For example, you'll find macros for the following values under `DCMD_CHR_LINESTATUS` in `<sys/dcmd_chr.h>`:

- Serial Port (DTR, RTS, CTS, DSR, RI, CD)
- Keyboard (Scroll/Caps/Num Lock, Shift, CTRL, ALT)
- Parallel Port (No Error, Selected, Paper Out, No Tack, Not Busy)

The value that's in the header is a “bitwise &” with the value in *data* to see if the value is high for that line.

Example 2: Cycling through Caps Lock, Num Lock, and Scroll Lock

In the following example, we open the device `/dev/kbd` and we start applying changes to the **Caps Lock**, **Scroll Lock**, and **Num Lock** properties.

The key lines in this example are the same as in the last example; they focus around the data variable. This value is just a simple integer value that's passed into the *devctl()* function. The data variable is assigned its values by simply performing a bitwise OR to the predefined values in the `</usr/include/sys/dcmd_chr.h>` header. Note the values used in the bitwise OR:

- `_CONCTL_NUM_CHG` (Console Control Num Lock Change) ORed together with `_CONCTL_NUM` (Console Control Num Lock) turns on **Num Lock**.
- `_CONCTL_NUM_CHG` on its own turns off **Num Lock**.

If <i>data</i> equals:	Num Lock is turned:
<code>_CONCTL_NUM_CHG _CONCTL_NUM</code>	ON
<code>_CONCTL_NUM_CHG</code>	OFF

This also applies for the other either/or values in the `<dcmd_chr.h>` header.

```

/* For "devctl()" */
#include <devctl.h>
#include <sys/dcmd_chr.h>

/* For "open()" */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* For Errors */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int data, fd, toggle = 1, error;

    /* Open the device we wish to manipulate. */
    if((fd = open ("/dev/kbd", O_RDONLY)) == -1)
    {
        fprintf(stderr,
            "Error with open() on /dev/kbd. Make sure it exists.\n");
        perror (NULL);
        exit(EXIT_FAILURE);
    }

    while(1)
    {
        switch(toggle)
        {
            case 1:
            {
                /*
                 Turn on Num Lock and make sure that
                 Caps and Scroll lock are turned off.
                */
                data = (_CONCTL_NUM_CHG | _CONCTL_NUM) | _CONCTL_CAPS_CHG |
                    _CONCTL_SCROLL_CHG;

```

```

        break;
    }
    case 2:
    {
        /*
         * Turn off Num Lock and now turn on Caps Lock
         * (Scroll lock is already off).
         */
        data = _CONCTL_NUM_CHG | (_CONCTL_CAPS_CHG | _CONCTL_CAPS);
        break;
    }
    case 3:
    {
        /*
         * Turn off Caps lock and turn on Scroll lock
         * (Num lock is already off).
         */
        data = _CONCTL_CAPS_CHG | (_CONCTL_SCROLL_CHG | _CONCTL_SCROLL);
        toggle = 0;
        break;
    }
}

/* Explanation below. */
if (error = devctl (fd, DCMD_CHR_SERCTL, &data,
                    sizeof(data), NULL))
{
    fprintf(stderr, "Error setting KBD: %s\n",
            strerror ( error ));
    exit(EXIT_FAILURE);
}

sleep(1);
toggle++;
}

return (1);
}

```

Here's a quick explanation of the above *devctl()* call:

```
devctl (fd, DCMD_CHR_SERCTL, &data, sizeof(data), NULL)
```

The first parameter, *fd*, is the file descriptor of the device that's being changed. The second parameter is the device class that's being changed. In this case, it's a character device *DCMD_CHR*, with a “subclass” of *_SERCTL*. The third parameter is the data variable; this is the ORed value.

Example 3: Duration example

In this code, *tcdropline()*, which is used to disconnect a communications line, uses *devctl()* (this is the actual source code, [tcdropline\(\)](#) (p. 3340) is a standard library function):

```

#include <termios.h>
#include <devctl.h>
#include <errno.h>
#include <sys/dcmd_chr.h>

int tcdropline(int fd, int duration) {
    int error;

    duration = ((duration ? duration : 300) << 16) |
        _SERCTL_DTR_CHG | 0;

    if(error = devctl(fd, DCMD_CHR_SERCTL, &duration,
                     sizeof duration, 0) == -1) {
        if(error == ENOSYS) {
            errno = ENOTTY;
        }
        return -1;
    }
}

```

```

    }    return 0;
}

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

When *devctl()* fails, the effect of the failed command depends on the device driver. The corresponding data might be transferred, partially transferred, or not transferred at all.

The *devctl()* function was originally part of the POSIX 1003.1d draft standard; but it was deprecated in the IEEE *Approved Draft 10* standard.

dhcctl*()

Functions for controlling DHCP

Synopsis:

```
#include <dhcctl/dhcctl.h>

dhcctl_status
dhcctl_initialize(void);

dhcctl_status
dhcctl_connect(dhcctl_handle *cxn, const char *host, int
port,
    dhcctl_handle auth);

dhcctl_status
dhcctl_wait_for_completion(dhcctl_handle object,
    dhcctl_status *status);

dhcctl_status
dhcctl_get_value(dhcctl_data_string *value, dhcctl_handle
object,
    const char *name);

dhcctl_status
dhcctl_get_boolean(int *value, dhcctl_handle object, const
char *name);

dhcctl_status
dhcctl_set_value(dhcctl_handle object, dhcctl_data_string
value,
    const char *name);

dhcctl_status
dhcctl_set_string_value(dhcctl_handle object, const char
*value,
    const char *name);

dhcctl_status
dhcctl_set_boolean_value(dhcctl_handle object, int value,
    const char *name);

dhcctl_status
dhcctl_set_int_value(dhcctl_handle object, int value,
    const char *name);

dhcctl_status
dhcctl_object_update(dhcctl_handle connection, dhcctl_handle
object);

dhcctl_status
dhcctl_object_refresh(dhcctl_handle connection, dhcctl_handle
object);

dhcctl_status
dhcctl_object_remove(dhcctl_handle connection, dhcctl_handle
object);
```



```

dhcpctl_status
dhcpctl_set_callback(dhcpctl_handle object, void *data,
    void (*function) (dhcpctl_handle, dhcpctl_status, void *));

dhcpctl_status
dhcpctl_new_authenticator(dhcpctl_handle *object, const char
    *name,
    const char *algorithm, const char *secret, unsigned
    secret_len);

dhcpctl_status
dhcpctl_new_object(dhcpctl_handle *object, dhcpctl_handle
    connection,
    const char *object_type);

dhcpctl_status
dhcpctl_open_object(dhcpctl_handle object, dhcpctl_handle
    connection,
    int flags);

isc_result_t
omapi_data_string_new(dhcpctl_data_string, *data, unsigned,
    int, length,
    const, char, *filename,, int, lineno);

isc_result_t
dhcpctl_data_string_dereference(dhcpctl_data_string *, const
    char *,
    int);

```

Description:

The *dhcpctl*()* set of functions provides an API that can be used to communicate with and manipulate a running ISC DHCP server. All functions return a value of *isc_result_t*. The return values reflects the result of operations to local data structures. If an operation fails on the server for any reason, then the error result will be returned through the second parameter of the *dhcpctl_wait_for_completion()* call.

- *dhcpctl_initialize()* sets up the data structures the library needs to do its work. This function must be called once before any other.
- *dhcpctl_connect()* opens a connection to the DHCP server at the given host and port. If an authenticator has been created for the connection, then it's given as the fourth argument. On a successful return, the address pointed at by the first argument will have a new connection object assigned to it. For example:

```
s = dhcpctl_connect(&cxn, "127.0.0.1", 7911, NULL);
```

connects to the DHCP server on the localhost via port 7911 (the standard OMAPI port). No authentication is used for the connection.

- *dhcpctl_wait_for_completion()* flushes a pending message to the server and waits for the response. The result of the request as processed on the server is returned via the second parameter:

```
s = dhcpctl_wait_for_completion(cxn, &wv);
```

```

if (s != ISC_R_SUCCESS)
    local_failure(s);
else if (wv != ISC_R_SUCCESS)
    server_failure(wc);

```

The call to *dhcpcctl_wait_for_completion()* won't return until the remote message processing completes or the connection to the server is lost.

- *dhcpcctl_get_value()* extracts a value of an attribute from the handle. The value can be of any length and is treated as a sequence of bytes. The handle must have been created first with *dhcpcctl_new_object()* and opened with *dhcpcctl_open_object()*. The value is returned via the *value* parameter. The last parameter is the name of attribute to retrieve:

```

dhcpcctl_data_string value = NULL;
dhcpcctl_handle lease;
time_t thetime;

s = dhcpcctl_get_value (&value, lease, "ends");
assert(s == ISC_R_SUCCESS && value->len == sizeof(thetime));
memcpy(&thetime, value->value, value->len);

```

- *dhcpcctl_get_boolean()* extracts a boolean-valued attribute from the object handle.
- The *dhcpcctl_set_value()*, *dhcpcctl_set_string_value()*, *dhcpcctl_set_boolean_value()*, and *dhcpcctl_set_int_value()* functions all set a value on the object handle.
- *dhcpcctl_object_update()* queues a request for all the changes made to the object handle be sent to the remote for processing. The changes made to the attributes on the handle will be applied to remote object if permitted.
- *dhcpcctl_object_refresh()* queues up a request for a fresh copy of all the attribute values to be sent from the remote to refresh the values in the local object handle.
- *dhcpcctl_object_remove()* queues a request for the removal on the server of the object referenced by the handle.
- The *dhcpcctl_set_callback()* function sets up a user-defined function to be called when an event completes on the given object handle. This is needed for asynchronous handling of events, versus the synchronous handling given by *dhcpcctl_wait_for_completion()*. When the function is called, the first parameter is the object the event arrived for, the second is the status of the message that was processed, the third is the same value as the second parameter given to *dhcpcctl_set_callback()*.
- The *dhcpcctl_new_authenticator()* creates a new authenticator object to be used for signing the messages that cross over the network. The *name*, *algorithm*, and *secret* values must all match what the server uses and are defined in its configuration file. The created object is returned through the first parameter and must be used as the fourth parameter to *dhcpcctl_connect()*. Note that the *secret* value must not be base64 encoded, which is different from how the value appears in the *dhcpcd.conf* file.
- *dhcpcctl_new_object()* creates a local handle for an object on the server. The *object_type* parameter is the ASCII name of the type of object being accessed. e.g.

Lease. This function only sets up local data structures; it doesn't queue any messages to be sent to the remote side — *dhcpctl_open_object()* does that.

- *dhcpctl_open_object()* builds and queues the request to the remote side. This function is used with handle created via *dhcpctl_new_object()*. The *flags* argument is a bitmask with the following values available for setting:

DHCPCTL_CREATE

If the object does not exist, then the remote will create it.

DHCPCTL_UPDATE

Update the object on the remote side using the attributes already set in the handle.

DHCPCTL_EXCL

Return an error if the object exists and DHCPCTL_CREATE was also specified.

- The *omapi_data_string_new()* function allocates a new *dhcpctl_data_string* object. The data string will be large enough to hold *length* bytes of data. The *file* and *lineno* arguments are the source file location the call is made from, typically by using the `__FILE__` and `__LINE__` macros or the MDL macro.
- *dhcpctl_data_string_dereference()* deallocates a data string created by *omapi_data_string_new()*. The memory for the object won't be freed until the last reference is released.

Examples:

The following program will connect to the DHCP server running on the local host and will get the details of the existing lease for IP address 10.0.0.101. It will then print out the time the lease is due to expire. Note that most error checking has been omitted for brevity.

```
#include <stdarg.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>

#include <isc/result.h>
#include <dhcpctl/dhcpctl.h>

int main (int argc, char **argv) {
    dhcpctl_data_string ipaddrstring = NULL;
    dhcpctl_data_string value = NULL;
    dhcpctl_handle connection = NULL;
    dhcpctl_handle lease = NULL;
    isc_result_t waitstatus;
    struct in_addr convaddr;
    time_t thetime;

    dhcpctl_initialize ();

    dhcpctl_connect (&connection, "127.0.0.1",
                    7911, 0);
```

```

    dhcpctl_new_object (&lease, connection,
                        "lease");

    memset (&ipaddrstring, 0, sizeof
            ipaddrstring);

    inet_pton(AF_INET, "10.0.0.101",
              &convaddr);

    omapi_data_string_new (&ipaddrstring,
                           4, MDL);
    memcpy(ipaddrstring->value, &convaddr.s_addr, 4);

    dhcpctl_set_value (lease, ipaddrstring,
                      "ip-address");

    dhcpctl_open_object (lease, connection, 0);

    dhcpctl_wait_for_completion (lease,
                                &waitstatus);
    if (waitstatus != ISC_R_SUCCESS) {
        /* server not authoritative */
        exit (0);
    }

    dhcpctl_data_string_dereference(&ipaddrstring,
                                    MDL);

    dhcpctl_get_value (&value, lease, "ends");
    memcpy(&thetime, value->value, value->len);
    dhcpctl_data_string_dereference(&value, MDL);
    fprintf (stdout, "ending time is %s",
            ctime(&thetime));
}

```

Classification:

NetBSD (p. 46)

Safety:	
Interrupt handler	No
Signal handler	No
Thread	No

Contributing author:

dhcpctl()* was written by Ted Lemon of Nominum, Inc. This preliminary documentation was written by James Brister of Nominum, Inc.

difftime()

Calculate the difference between two times

Synopsis:

```
#include <time.h>

double difftime( time_t time1,
                 time_t time0 );
```

Arguments:

time1, time0

The times to compare, expressed as `time_t` objects.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *difftime()* function calculates the difference between the calendar times specified by *time1* and *time0*:

time1 - *time0*

Returns:

The difference between the two times (in seconds).

Examples:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void compute( void )
{
    int i, j;

    for( i = 1; i <= 20; i++ ) {
        for( j = 1; j <= 20; j++ ) {
            printf( "%3d ", i * j );
        }
        printf( "\n" );
    }
}

int main( void )
{
```

```
time_t start_time, end_time;

start_time = time( NULL );
compute();
end_time = time( NULL );
printf( "Elapsed time: %f seconds\n",
        difftime( end_time, start_time ) );

return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dircntl()

Control an open directory

Synopsis:

```
#include <dirent.h>

int dircntl( DIR * dir,
            int cmd,
            ... );
```

Arguments:

dir

Provide control for this directory.

cmd

At least the following values are defined in `<dirent.h>`:

- `D_GETFLAG` — retrieve the flags associated with the directory referenced by *dir*. For more information, see “[Flag values](#) (p. 419),” below.
- `D_SETFLAG` — set the flags associated with the directory referenced by *dir* to the value given as an additional argument. The new value can be any combination of the flags described in “[Flag values](#) (p. 419),” below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dircntl()* function provides control over the open directory referenced by the *dir* argument. This function behaves in a manner similar to the file control function, *fcntl()* (p. 623).

Flag values**D_FLAG_FILTER**

Filter out duplicate name entries that may occur due to the union filesystem during a *readdir()* (p. 2514) operation.

D_FLAG_STAT

Indicate to servers that they should attempt to return extra [stat\(\)](#) (p. 3164) information as part of the [readdir\(\)](#) (p. 2514) operation.

Returns:

The return value depends on the value of *cmd*:

D_GETFLAG

The flags associated with the directory.

D_SETFLAG

0.

Any other value

-1 ([errno](#) (p. 539) is set to ENOSYS).

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main(int argc, char **argv) {
    DIR *dp;
    int ret;

    if(!(dp = opendir("/"))) {
        exit(EXIT_FAILURE);
    }

    /* Display the flags that are set on the
       directory by default*/
    if((ret = dircntl(dp, D_GETFLAG)) == -1) {
        exit(EXIT_FAILURE);
    }

    if(ret & D_FLAG_FILTER) {
        printf("Directory names are filtered\n");
    } else {
        printf("Directory names are not filtered\n");
    }

    if(ret & D_FLAG_STAT) {
        printf("Servers asked for extra stat information\n");
    } else {
        printf("Servers not asked for extra stat information\n");
    }

    closedir(dp);
    return 0;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dirent, dirent64

Data structure for a directory entry

Synopsis:

```
#include <dirent.h>

struct dirent {
#if _FILE_OFFSET_BITS - 0 == 64
    ino_t      d_ino;      /* File serial number. */
    off_t      d_offset;
#elif !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
    ino_t      d_ino;      /* File serial number. */
    ino_t      d_ino_hi;
    off_t      d_offset;
    off_t      d_offset_hi;
#elif defined(__BIGENDIAN__)
    ino_t      d_ino_hi;
    ino_t      d_ino;      /* File serial number. */
    off_t      d_offset_hi;
    off_t      d_offset;
#else
    #error endian not configured for system
#endif
    #error _FILE_OFFSET_BITS value is unsupported
#endif
    int16_t     d_reclen;
    int16_t     d_namelen;
    char        d_name[1];
};

struct dirent64 {
    ino64_t     d_ino;
    off64_t     d_offset;
    _Int16t     d_reclen;
    _Int16t     d_namelen;
    char        d_name[1];
};
```

Description:

The `dirent` structure describes an entry in a directory. The members include:

d_ino

A mountpoint-unique file serial number. This serial number is often used in various disk-checking utilities for such operations as determining infinite-loop directory links. (Note that the inode value cannot be zero, which would indicate that the inode represents an *unused* entry.)

d_offset

In some filesystems, this member identifies the directory entry itself; in others, it's the offset of the *next* directory entry. For a disk-based filesystem, this value might be the actual offset into the on-disk directory structure.

d_reclen

The size of this directory entry *and any other associated information* (such as an optional `struct stat` structure appended to the `struct dirent` entry).

d_namelen

The size of the `d_name` member. Since the size is calculated using [strlen\(\)](#) (p. 3213), the `\0` string terminator, which must be present, *isn't* counted.

d_name

The actual name of that directory entry.

These structures include space only for the first four bytes of the pathname. If you create an instance of this structure, you must provide space for the name, including the terminating null character:



```
struct dirent *entry;  
entry = malloc( offsetof(struct dirent, d_name) + NAME_MAX + 1 );
```

or:

```
struct {  
    struct dirent ent;  
    char namebuf[NAME_MAX + 1 + offsetof(struct dirent, d_name) -  
                sizeof( struct dirent)];  
} entry
```

Classification:

`dirent` is [POSIX 1003.1](#) (p. 46); `dirent64` is [Large-file support](#) (p. 46)

dirname()

Find the parent directory part of a file pathname

Synopsis:

```
#include <libgen.h>

char *dirname( char *path );
```

Arguments:

path

The string to parse.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dirname()* function takes a pointer to a character string that contains a pathname, and returns a pointer to a string that's a pathname of the parent directory of that file. Trailing “/” characters in the path aren't counted as part of the path.

If the path *doesn't* contain a “/” character, or *path* is a NULL pointer or points to an empty string, then *dirname()* function returns a pointer to the string “.” (dot).

Together the *dirname()* and [basename\(\)](#) (p. 202) functions yield a complete pathname. The expression *dirname(path)* obtains the pathname of the directory where *basename(path)* is found.



The *dirname()* function might modify the string pointed to by *path*, and can return a pointer to static storage.

Returns:

A pointer to a string that's the parent directory of *path*. If *path* is a NULL pointer or points to an empty string, a pointer to a string “.” is returned.

Examples:

String input	String output
"/usr/lib"	"/usr"
"/usr/"	"/"
"/"	"/"
"."	"."
". ."	". ."

The following code fragment reads a pathname, changes the current working directory to the parent directory, and opens the file:

```
char path[BUFF_SIZE], *pathcopy;
int fd;

fgets(path, BUFF_SIZE, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

dispatch_block()

Block while waiting for an event

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_context_t * dispatch_block
    ( dispatch_context_t * ctp );
```

Arguments:

ctp

A pointer to a `dispatch_context_t` structure that defines the dispatch context.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dispatch_block()* function blocks while waiting for an event (e.g. message or signal) that's registered using one of the attach functions, [message_attach\(\)](#) (p. 1624), [pulse_attach\(\)](#) (p. 2442), [resmgr_attach\(\)](#) (p. 2580), or [select_attach\(\)](#) (p. 2796). (The *sigwait_attach()* function isn't currently implemented.)

If the type of blocking is:	<i>dispatch_block()</i> does a:
message (resmgr, message, select)	MsgReceive() (p. 1760)
signal	SignalWaitinfo() (p. 2997)

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

A dispatch context that's passed in by [dispatch_context_alloc\(\)](#) (p. 429). or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors can occur when the blocking kernel call returns with an error, for example, due to the delivery of a signal.



In the case of a timeout, a valid *ctp* is returned, but either the *ctp->message_context.rcvid* or *ctp->sigwait_context.signo* is set to -1.

If a non-NULL context pointer is returned, it could be different from the one passed in, as it's possible for the *ctp* to be reallocated to a larger size. In this case, the old *ctp* is no longer valid. However, if NULL is returned (for example, because a signal interrupted the *MsgReceive()*), the old context pointer is still valid. Typically, a resource manager would target signals to a thread dedicated to handling signals. However, if a signal can be targeted to the thread doing *dispatch_block()*, you could use the following code in this situation:

```
dispatch_context_t  *ctp, *new_ctp;
ctp = dispatch_context_alloc( ... );
while (1) {
    new_ctp = dispatch_block( ctp );
    if ( new_ctp ) {
        ctp = new_ctp
    }
    else {
        /* handle the error condition */
        ...
    }
}
```

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers.

EINTR

The call was interrupted by a signal.

EINVAL

Invalid arguments passed to *dispatch_block()*.

ENOMEM

Insufficient memory to allocate internal data structures.

See also the error constants returned in [MsgReceive\(\)](#) (p. 1760) and [SignalWaitinfo\(\)](#) (p. 2997).

Examples:

```
#include <sys/dispatch.h>

int main( int argc, char **argv ) {
    dispatch_context_t  *ctp;
    ...
}
```

```
for(;;) {  
    if( ctp = dispatch_block( ctp ) ) {  
        dispatch_handler( ctp );  
    }  
}
```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

dispatch_context_alloc()

Return a dispatch context

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_context_t * dispatch_context_alloc
( dispatch_t * dpp );
```

Arguments:

dpp

A dispatch handle created by a successful call to [dispatch_create\(\)](#) (p. 433).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *dispatch_context_alloc()* function returns a dispatch context pointer. The function is passed in the handle *dpp* from [dispatch_create\(\)](#) (p. 433). The dispatch context is used by dispatch to do its work. It's passed as an argument to [dispatch_block\(\)](#) (p. 426) and [dispatch_handler\(\)](#) (p. 440).



The *dispatch_context_alloc()* function fails if you haven't attached any events to dispatch yet (e.g. you didn't call [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), or [select_attach\(\)](#) (p. 2796)). The dispatch library can't allocate a proper context until it knows what kind of events you want to block.

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

A pointer to a dispatch context, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

No events were attached.

ENOMEM

Insufficient memory to allocate context.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    dispatch_context_t  *ctp;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    &vellip;

    ctp = dispatch_context_alloc( dpp );

    &vellip;

    return EXIT_SUCCESS;
}
```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dispatch_context_free()

Free a dispatch context

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void dispatch_context_free(
    dispatch_context_t * ctp );
```

Arguments:

ctp

A pointer to a `dispatch_context_t` structure that was allocated by [dispatch_context_alloc\(\)](#) (p. 429).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dispatch_context_free()` function frees the given dispatch context.

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    dispatch_context_t *ctp;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate
                  dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    ...

    ctp = dispatch_context_alloc( dpp );

    ...
}
```

```
    dispatch_context_free ( ctp );  
    return EXIT_SUCCESS;  
}
```

See [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387) for examples using the dispatch interface.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dispatch_create()

Allocate a dispatch handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_t *dispatch_create( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dispatch_create()* function allocates and initializes a dispatch handle. The attach functions are:

- [message_attach\(\)](#) (p. 1624)
- [pulse_attach\(\)](#) (p. 2442)
- [resmgr_attach\(\)](#) (p. 2580)
- [select_attach\(\)](#) (p. 2796)

If you wish, you can do a [resmgr_attach\(\)](#) (p. 2580) with a NULL path. This has the effect of initializing dispatch to receive messages and creates the channel among other things.



A channel is created only when you first attach something that requires a channel (indicating you will block receiving messages).

If you want to create the channel yourself (e.g. in order to use [name_attach\(\)](#) (p. 1842)), call [dispatch_create_channel\(\)](#) (p. 436) instead of *dispatch_create()*.

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

A handle to a dispatch structure, or NULL if an error occurs.



The dispatch structure, `dispatch_t`, is an opaque data type; you can't access its contents directly.

Errors:

ENOMEM

Insufficient memory to allocate context.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <fcntl.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int my_func( select_context_t *ctp, int fd,
             unsigned flags, void *handle ) {
    int i, c;

    /* Do some useful stuff with data */
    i = read( fd, &c, 1 );
    fprintf( stderr, "activity on fd %d: read char %c,
                  return code %d\n", fd, c, i );
}

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    dispatch_context_t *ctp;
    select_attr_t    attr;
    int              fd, fd2;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                  dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    if( argc < 2 || (fd = open( argv[1],
                               O_RDWR | O_NONBLOCK )) == -1 ) {
        return EXIT_FAILURE;
    }

    if( argc < 3 || (fd2 = open( argv[2],
                                 O_RDWR | O_NONBLOCK )) == -1 ) {
        return EXIT_FAILURE;
    }

    select_attach( dpp, &attr, fd,
                   SELECT_FLAG_READ | SELECT_FLAG_REARM, my_func, NULL );
    select_attach( dpp, &attr, fd2,
                   SELECT_FLAG_READ | SELECT_FLAG_REARM, my_func, NULL );
    ctp = dispatch_context_alloc( dpp );

    for(;;) {
        if( ctp = dispatch_block( ctp ) ) {
            dispatch_handler( ctp );
        }
    }
    return EXIT_SUCCESS;
}
```

For more examples using the dispatch interface, see [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dispatch_create_channel()

Allocate a dispatch handle, specifying a channel ID

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_t *dispatch_create_channel( int chid,
                                     unsigned reserved );
```

Arguments:

chid

The ID of the channel to use for the dispatch layer.

reserved

Reserved; specify 0 for this argument.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dispatch_create_channel()* function allocates and initializes a dispatch handle. The attach functions are:

- [message_attach\(\)](#) (p. 1624)
- [pulse_attach\(\)](#) (p. 2442)
- [resmgr_attach\(\)](#) (p. 2580)
- [select_attach\(\)](#) (p. 2796)

If you wish, you can do a [resmgr_attach\(\)](#) (p. 2580) with a NULL path. This has the effect of initializing dispatch to receive messages, among other things.

This function is similar to [dispatch_create\(\)](#) (p. 433), but lets you specify a channel for the dispatch to use. It also lets you specify channel flags for *name_attach()*.

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

A handle to a dispatch structure, or NULL if an error occurs.



The dispatch structure, `dispatch_t`, is an opaque data type; you can't access its contents directly.

Errors:**ENOMEM**

Insufficient memory to allocate context.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dispatch_destroy()

Destroy a dispatch handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int dispatch_destroy( dispatch_t *dpp );
```

Arguments:

dpp

A dispatch handle created by a successful call to [dispatch_create\(\)](#) (p. 433).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *dispatch_destroy()* destroys the given dispatch handle.

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The dispatch handle, *dpp*, is invalid.

Examples:

```

#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int destroyed;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    ...

    if ( (destroyed = dispatch_destroy ( dpp )) == -1 ) {
        fprintf ( stderr, "Dispatch wasn't destroyed, \
                bad dispatch handle %d.\n", dpp);
        return EXIT_FAILURE;
    }
    /* else dispatch was destroyed */

    ...
    return EXIT_SUCCESS;
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dispatch_handler()

Handle events received by dispatch_block()

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int dispatch_handler( dispatch_context_t * ctp );
```

Arguments:

ctp

A pointer to a `dispatch_context_t` structure that was allocated by [dispatch_context_alloc\(\)](#) (p. 429).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *dispatch_handler()* function handles events received by [dispatch_block\(\)](#) (p. 426). This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Depending on the blocking type, *dispatch_handler()* does one of the following:

- Calls the [message_*](#) (p. 1624) subsystem. A search is made (based upon the message type or pulse code) for a matching function (that was attached with [message_attach\(\)](#) (p. 1624) or [pulse_attach\(\)](#) (p. 2442)). If a match is found, the attached function is called.
- If the message type is in the range handled by the resource manager (e.g. I/O messages) and pathnames were attached using [resmgr_attach\(\)](#) (p. 2580), then the [resmgr_*](#) (p. 2580) subsystem is called and handles the resource manager messages.
- If a pulse is received, it may be dispatched to the [resmgr_*](#) subsystem if it's one of the codes (unblock and disconnect pulses) handled by the resource manager. If a [select_attach\(\)](#) (p. 2796) was done and the pulse matches the one used by [select_attach\(\)](#), then the [select_*](#) subsystem is called and dispatches that event.

- If a message is received, and no matching handler is found for that message type, [MsgError\(\)](#) (p. 1743) returns ENOSYS to the sender.
- If a [SignalWaitinfo\(\)](#) (p. 2997) blocking type is used, then a search is made based upon the signal number for a matching function attached by the program (using the [sigwait_attach\(\)](#) function, not currently implemented). If a match is found, that function is called.

Returns:

0

Success.

-1

One of the following occurred:

- The message was a [_PULSE_CODE_THREADDEATH](#) (p. 281) pulse message (see [ChannelCreate\(\)](#)) for which there's no default handler function.
- The message length was less than 2 bytes. A 2-byte message type is required at the beginning of the message so that a handler function can be found or identified.
- The message wasn't in native endian format and there were no handler functions that specified MSG_FLAG_CROSS_ENDIAN on this range, even though a handler was registered for it using [message_attach\(\)](#) (p. 1624). The MSG_FLAG_CROSS_ENDIAN flag wasn't given to [message_attach\(\)](#).
- A handler was found for the message, but the handler determined that there was a problem.

In any case, if the message wasn't a pulse, then the client will be replied to with an appropriate [errno](#) (p. 539).

Examples:

```
#include <stdlib.h>
#include <sys/dispatch.h>

int main( int argc, char **argv ) {
    dispatch_context_t *ctp;

    ...

    for(;;) {
        if( ctp = dispatch_block( ctp ) ) {
            dispatch_handler( ctp );
        }
    }
    return EXIT_SUCCESS;
}
```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

This function might or might not be a cancellation point, depending on the implementation of the handler.

dispatch_timeout()

Set a timeout

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int dispatch_timeout( dispatch_t *dpp,
                     struct timespec *reltime );
```

Arguments:

dpp

A dispatch handle created by a successful call to [dispatch_create\(\)](#) (p. 433).

reltime

A pointer to a [timespec](#) (p. 3473) structure that specifies the relative time of the timeout.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *dispatch_timeout()* sets a timeout that's used when blocking with [dispatch_block\(\)](#) (p. 426).

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

0

Success.

-1

An error occurred.

Examples:

```

#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    struct timespec  time_out;
    int              timedout;
    time_out.tv_sec = 1;
    time_out.tv_nsec = 2;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    ...

    if ( (timedout = dispatch_timeout ( dpp, &time_out ))
        == -1 ) {
        fprintf ( stderr, "Couldn't set timeout );
        return EXIT_FAILURE;
    }
    /* else successful timeout set */

    ...
    return EXIT_SUCCESS;
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dispatch_unblock()

Unblock all of the threads that are blocked on a dispatch handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void dispatch_unblock( dispatch_context_t * ctp );
```

Arguments:

ctp

A pointer to a `dispatch_context_t` structure that defines the dispatch context.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

This routine tries to unblock all of the threads that are blocked on the given dispatch handle. You should use this function in the thread pool structure as the unblock function pointer so that [thread_pool_control\(\)](#) (p. 3385) will behave properly.

Currently, this function unblocks only channel resources.

This function is part of the dispatch layer of a resource manager. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Examples:

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

div()

Calculate a quotient and remainder

Synopsis:

```
#include <stdlib.h>

div_t div( int numer,
           int denom );
```

Arguments:

numer

The numerator in the division.

denom

The denominator.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `div()` function calculates the quotient and remainder of the division of *numer* by *denom*.

Returns:

A `div_t` structure containing the quotient and remainder:

```
typedef struct {
    int quot;      /* quotient */
    int rem;       /* remainder */
} div_t;
```

Examples:

```
#include <stdio.h>
#include <stdlib.h>

void print_time( int seconds )
{
    div_t min_sec;

    min_sec = div( seconds, 60 );
    printf( "It took %d minutes and %d seconds\n",
```

```
        min_sec.quot, min_sec.rem );
}

int main( void )
{
    print_time( 130 );

    return EXIT_SUCCESS;
}
```

produces the output:

```
It took 2 minutes and 10 seconds
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

dladdr()

Translate an address to symbolic information

Synopsis:

```
#include <dlfcn.h>

int dladdr( void *address,
            DL_info *dli );
```

Arguments:

address

The address for which you want symbolic information.

dli

A pointer to a `DL_info` structure where the function can store the symbolic information. Your application must allocate the space for this structure; *dladdr()* fills in the members, based on the specified address.

The `DL_info` structure includes the following members:

- `const char * dli_fname` — a pointer to the full path of the object containing *address*.
- `void *dli_fbase` — the base address of the object containing *address*.
- `const char *dli_sname` — a pointer to the symbol name nearest the specified *address*. This symbol is either at *address*, or is the nearest symbol with a lower address.
- `void *dli_saddr` — the actual address of the *dli_sname* symbol.

If *dladdr()* can't find a symbol that describes the specified *address*, the function sets *dli_sname* and *dli_saddr* to `NULL`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dladdr()* function determines whether the specified *address* is located within one of the objects that make up the calling application's address space.



The `dladdr()` function is available only to dynamically linked processes.

Returns:

0 if the specified *address* can't be matched, or nonzero if it could be matched.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The `DL_info` pointers may become invalid if objects are removed via [dlclose\(\)](#) (p. 451).

There's no way to determine which symbol you'll get if multiple symbols are mapped to the same address.

dldclose()

Close a shared object

Synopsis:

```
#include <dldfcn.h>

int dldclose( void *handle );
```

Arguments:

handle

A handle for a shared object, returned by [dldopen\(\)](#) (p. 454).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *dldclose()* function disassociates a shared object opened by [dldopen\(\)](#) (p. 454) from the calling process. An object's symbols are no longer available after it's been closed with *dldclose()*. All objects loaded as a result of the closed objects dependencies are also closed.

The *handle* argument is the value returned by a previous call to *dldopen()*.



The *dldclose()* function is available only to dynamically linked processes.

Returns:

0 for success, or a nonzero value if an error occurs.

Errors:

If an error occurs, more detailed diagnostic information is available from [dlderror\(\)](#) (p. 453).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

An object won't be removed from the address space until all references to that object (via *dlopen()* or dependencies from other objects) have been closed.

Referencing a symbol in a closed object can cause undefined behavior.

derror()

Get dynamic loading diagnostic information

Synopsis:

```
#include <dlfcn.h>

char *derror( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *derror()* function returns a **NULL**-terminated string (with no trailing newline) describing the last error that occurred during a call to one of the *dl*()* functions. If no errors have occurred, *derror()* returns **NULL**.



The *dlopen()* function is available only to dynamically linked processes.

Returns:

A pointer to an error description, or **NULL**.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

dlopen()

Gain access to an executable object file

Synopsis:

```
#include <dlfcn.h>

void * dlopen( const char * pathname,
               int mode );
```

Arguments:

pathname

NULL, or the path to the executable object file that you want to access.

mode

Flags that control how *dlopen()* operates. POSIX defines these bits:

- RTLD_LAZY
- RTLD_NOW
- RTLD_GLOBAL
- RTLD_LOCAL

The following bits are Unix extensions:

- RTLD_NOLOAD
- RTLD_GROUP
- RTLD_WORLD
- RTLD_NODELETE

The following bits are QNX Neutrino extensions:

- RTLD_NOSHARE
- RTLD_LAZYLOAD

For more information, see “[The mode](#) (p. 456),” below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *dlopen()* function gives you direct access to the dynamic linking facilities by making the executable object file specified in *pathname* available to the calling process. It returns a handle that you can use in subsequent calls to *dlsym()* (p. 461) and *dlclose()* (p. 451).



- In order to successfully call this function, your process must have the PROC_MGR_AID_PROT_EXEC ability enabled. For more information, see *procmgr_ability()* (p. 2141).
- If your process has any privileged abilities enabled, then the executable file you're trying to execute must be trusted. For more information, see the description of PROT_EXEC in the entry for *mmap()* (p. 1656), as well as the entry for *pathtrust* in the *Utilities Reference*.
- The *dlopen()* function is available only to a dynamically-linked process. A statically-linked process (one where `libc` is linked statically) can't call *dlopen()* because a statically-linked executable:
 - doesn't export any of its symbols
 - can't export the required structure for libraries to link against
 - can't fill structures at startup needed to load subsequent shared objects

Any dependencies recorded within *pathname* are loaded as part of the *dlopen()* call. These dependencies are searched in load-order to locate any additional dependencies. This process continues until all of the dependencies for *pathname* have been satisfied. This dependency tree is called a *group*.

If *pathname* is `NULL`, *dlopen()* provides a handle to the running process's global symbol object. This provides access to the symbols from the original program image file, the dependencies it loaded at startup, plus any objects opened with *dlopen()* calls using the `RTLD_GLOBAL` flag. This set of symbols can change dynamically if the application subsequently calls *dlopen()* using `RTLD_GLOBAL`.

You can use *dlopen()* any number of times to open objects whose names resolve to the same absolute or relative path name; the object is loaded into the process's address space only once.

In order to find the shared objects, *dlopen()* searches the following, in this order:

- the runtime library search path that was set using the `-rpath` option to `ld` (see the *Utilities Reference*) when the binary was linked
- directories specified by the **`LD_LIBRARY_PATH`** environment variable
- directories specified by the `_CS_LIBPATH` configuration string

Note that ***LD_LIBRARY_PATH*** is ignored if the binary is `setuid` and the effective user ID isn't the same as the actual ID of the user running the binary. This is done for security purposes.

The above directories are set as follows:

- The ***LD_LIBRARY_PATH*** is generally set up by a startup script, either in the boot image or by a secondary script. It isn't part of any default environment.



- ***_CS_LIBPATH*** is populated by the kernel, and the default value is based on the ***LD_LIBRARY_PATH*** value of the `procnto` command line in the boot image. Note that you can use `getconf` to inspect this value and `setconf` to set it. For example:

```
setconf _CS_LIBPATH 'getconf _CS_LIBPATH':/new/path
```

When loading shared objects, the application should open a specific version instead of relying on the version pointed to by a symbolic link.

The *mode*

The *mode* argument indicates how `dlopen()` operates on *pathname* when handling relocations, and controls the visibility of symbols found in *pathname* and its dependencies.

The *mode* argument is a bitwise-OR of the constants described below. Note that the relocation and visibility constants are mutually exclusive.

Relocation

When you load an object by calling `dlopen()`, the object may contain references to symbols whose addresses aren't known until the object has been loaded; these references must be relocated before accessing the symbols. The *mode* controls when relocations take place, and can be one of:

RTLD_LAZY

(QNX Neutrino 6.5.0 and later) References to data symbols are relocated when the object is loaded. References to functions aren't relocated until that function is invoked. This improves performance by preventing unnecessary relocations.

RTLD_NOW

All references are relocated when the object is loaded. This may waste cycles if relocations are performed for functions that never get called, but this behavior could be useful for applications that need to know that all symbols referenced during execution are available as soon as the object is loaded.

Visibility

The following *mode* bits determine the scope of visibility for symbols loaded with *dlopen()*:

RTLD_GLOBAL

Make the object's global symbols available to any other object that's opened later with *RTLD_WORLD*. Symbol lookup using *dlopen(0, mode)* and an associated *dlsym()* (p. 461) are also able to find the object's symbols.

RTLD_LOCAL

Make the object's global symbols available only to objects in the same group.

RTLD_LAZYLOAD

Open the shared object and form its resolution scope by appending its immediate dependencies. This is different from the normal resolution scope, which is formed by appending the whole dependency tree in breadth-first order. For more information, see “Lazy loading” in the Compiling and Debugging chapter of the QNX Neutrino *Programmer's Guide*.

The program's image and any objects loaded at program startup have a *mode* of *RTLD_GLOBAL*; the default *mode* for objects acquired with *dlopen()* is *RTLD_LOCAL*. A local object may be part of the dependencies for more than one group; any object with a *RTLD_LOCAL mode* referenced as a dependency of an object with a *RTLD_GLOBAL mode* is promoted to *RTLD_GLOBAL*.

Objects loaded with *dlopen()* that require relocations against global symbols can reference the symbols in any *RTLD_GLOBAL* object.

Symbol scope

You can OR the *mode* with the following values to affect the symbol scope:

RTLD_GROUP

Only symbols from the associated group are available. All dependencies between group members must be satisfied by the objects in the group.

RTLD_WORLD

Only symbols from *RTLD_GLOBAL* objects are available.

If you don't specify either of these values, *dlopen()* uses *RTLD_WORLD* | *RTLD_GROUP*.



If you specify *RTLD_WORLD* without *RTLD_GROUP*, *dlopen()* doesn't load any of the DLL's dependencies.

Other flags

The following flags provide additional capabilities:

RTLD_NODELETE

Don't delete the specified object from the address space as part of a call to [dlclose\(\)](#) (p. 451).

RTLD_NOLOAD

Don't load the specified object, but return a valid handle if the object already exists as part of the process address space. You can specify addition modes, which are ORed with the present mode of the object and its dependencies. This flag gives you a means of querying the presence or promoting the modes of an existing dependency.

RTLD_NOSHARE

Don't share the specified object. This flag forces the loading of multiple instances of libraries.

Symbol resolution

When resolving the symbols in the shared object, the runtime linker searches for them in the dynamic symbol table using the following order:

By default:

1. main executable
2. the shared object being loaded
3. objects specified by the **LD_PRELOAD** environment variable
4. all other loaded shared objects that were loaded with the [RTLD_GLOBAL](#) (p. ?) flag

When -Bsymbolic is specified:

1. the shared object being loaded
2. main executable
3. objects specified by the **LD_PRELOAD** environment variable
4. all other loaded shared objects that were loaded with the [RTLD_GLOBAL](#) (p. ?) flag

For executables, the dynamic symbol table typically contains only those symbols that are known to be needed by any shared libraries. This is determined by the linker when the executable is linked against a shared library.

Since you don't link your executable against a shared object that you load with [dlopen\(\)](#), the linker can't determine which executable symbols need to be made available to the shared object.

If your shared object needs to resolve symbols in the executable, then you may force the linker to make *all* of the symbols in the executable available for dynamic linking by specifying the `-E` linker option. For example:

```
gcc -Vgcc_ntox86 -Wl,-E -o main main.o
```

Shared objects always place all their symbols in dynamic symbol tables, so this option isn't needed when linking a shared object.

Returns:

A handle to the object, or `NULL` if an error occurs.



Don't interpret the value of this handle in any way. For example, if you open the same object repeatedly, don't assume that `dlopen()` returns the same handle.

Errors:

If an error occurs, more detailed diagnostic information is available from [`dlderror\(\)`](#) (p. 453).

Environment variables:

`DL_DEBUG`

If this environment variable is set, the shared library loader displays debugging information about the libraries as they're opened. The value can be a comma-separated list of the following:

- `all` — display all debug messages.
- `help` — display a help message, and then exit.
- `reloc` — display relocation processing messages.
- `libs` — display information about shared objects being opened.
- `statistics` — display runtime linker statistics.
- `lazyload` — print lazy-load debug messages.
- `debug` — print various runtime linker debug messages.

A value of 1 (one) is the same as `all`.

`LD_BIND_NOW`

Affects lazy-load dependencies due to full symbol resolution. Typically, it forces the loading of all lazy-load dependencies (until all symbols have been resolved).

`LD_DEBUG`

A synonym for **DL_DEBUG**. If you set both **DL_DEBUG** and **LD_DEBUG**, then **DL_DEBUG** takes precedence.

LD_DEBUG_OUTPUT

The name of a file in which the dynamic linker writes its output. By default, output is written to *stderr*.



For security reasons, the use of **LD_DEBUG_OUTPUT** with *setuid* binaries is disabled.

LD_LIBRARY_PATH

A colon-separated list of directories to search for shared libraries.

LD_PRELOAD

A list of full paths to the shared libraries on an ELF system that you want to load before loading other libraries. Use a colon (:) to separate the libraries in this list. You can use this environment variable to add or change functionality when you run a program. For *setuid* or *setgid* ELF binaries, only libraries in the standard search directories that are also *setuid* will be loaded.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Some symbols defined in executables or shared objects might not be available to the runtime linker. The symbol table created by *ld* for use by the runtime linker might contain a subset of the symbols defined in the object.

dlsym()

Get the address of a symbol in a shared object

Synopsis:

```
#include <dlfcn.h>

void* dlsym( void* handle,
             const char* name );
```

Arguments:

handle

Either a handle for a shared object, returned by [dlopen\(\)](#) (p. 454), or either of the special flags, RTLD_NEXT or RTLD_DEFAULT.

name

The name of the symbol that you want to find in the shared object.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *dlsym()* function lets a process obtain the address of the symbol specified by *name* defined in a shared object.



The *dlsym()* function is available only to dynamically linked processes.

If *handle* is a handle returned by *dlopen()*, you must not have closed that shared object by calling [dlclose\(\)](#) (p. 451). The *dlsym()* functions also searches for the named symbol in the objects loaded as part of the dependencies for that object.

If *handle* is RTLD_NEXT, *dlsym()* searches the objects loaded after the object calling *dlsym()*.

If *handle* is RTLD_DEFAULT, *dlsym()* searches all objects in the current process, in load-order.

In the case of both RTLD_NEXT and RTLD_DEFAULT, if the objects being searched were loaded with *dlopen()*, *dlsym()* searches the object only if the caller is part of the same dependency hierarchy, or if the object was loaded with global search access (using the RTLD_GLOBAL mode).

Returns:

A pointer to the named symbol for success, or NULL if an error occurs.

Errors:

If an error occurs, more detailed diagnostic information is available from *dlderror()* (p. 453).

Examples:

Use *dlsym()* to find a function pointer and a pointer to a global variable in a shared library:

```
typedef int (*foofunc)( int );

void* handle;
int* some_global_int;
foofunc brain;

/* Open a shared library. */
handle = dlopen( "/usr/nto/x86/lib/libfoo.so.1", RTLD_NOW );

/* Find the address of a function and a global integer. */
brain = (foofunc)dlsym( handle, "takeover_world" );
some_global_int = (int* )dlsym( handle, "my_global_int" );

/* Invoke the function and print the int. */
x = (*brain)( 5 );
printf( "that global is %d\n", *some_global_int );
```

Check to see if a function is defined, and call it if it is:

```
typedef int (*funcptr)( void );

funcptr funk = NULL;

funk = (funcptr)dlsym( RTLD_DEFAULT, "get_funky" );
if( funk != NULL ) {
    (*funk)();
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Function pointers are a pain; use `typedefs` to help preserve your sanity.

dn_comp()

Compress an Internet domain name

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp( const char * exp_dn,
             u_char * comp_dn,
             int length,
             u_char ** dnptrs,
             u_char ** lastdnptr );
```

Arguments:

exp_dn

The Internet domain name you want to compress.

comp_dn

A buffer where the function can store the compressed name.

length

The size of the array that *comp_dn* points to.

dnptrs

NULL, or an array of pointers to previously compressed names in the current message; see below.

lastdnptr

NULL, or the limit of the array specified by *dnptrs*.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *dn_comp()* routine is a low-level routine used by [res_query\(\)](#) (p. 2569) to compress an Internet domain name. This routine compresses the domain name *exp_dn* and stores it in *comp_dn*.

The compression uses an array of pointers, *dnptrs*, to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with **NULL**. The limit to the array is specified by *lastdnptr*. As a side effect, *dn_comp()* updates the list of pointers for labels inserted into the message as the name is compressed. If *dnptrs* is **NULL**, names aren't compressed. If *lastdnptr* is **NULL**, the list of labels isn't updated.

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

The size of the compressed domain name, in bytes, or -1 if an error occurs.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

dn_expand()

Expand a compressed Internet domain name

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand( const u_char * msg,
               const u_char * eomorig,
               const u_char * comp_dn,
               char * exp_dn,
               int length );
```

Arguments:

msg

A pointer to the beginning of the message that contains the compressed name.

eomorig

A pointer to the first location after the message.

comp_dn

The compressed name that you want to expand.

exp_dn

A buffer where the function can store the expanded name.

length

The length of the array specified by *exp_dn*.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *dn_expand()* function is a low-level routine used by [res_query\(\)](#) (p. 2569) to expand the compressed domain name, *comp_dn*, to a full domain name.

The compressed name is contained in a query or reply message.

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

The size of the *compressed* domain name (not the expanded name), in bytes, or -1 if an error occurs.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

drand48()

Generate a pseudo-random double

Synopsis:

```
#include <stdlib.h>

double drand48( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *drand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative **double** uniformly distributed over the interval [0.0, 1.0].

Call one of [lcong48\(\)](#) (p. 1505), [seed48\(\)](#) (p. 2787), or [srand48\(\)](#) (p. 3157) to initialize the random-number generator before calling *drand48()*, [lrand48\(\)](#) (p. 1551), or [mrand48\(\)](#) (p. 1730).

The [erand48\(\)](#) (p. 531) function is a thread-safe version of *drand48()*.

Returns:

A pseudo-random double.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

ds_clear()

Delete a data server variable

Synopsis:

```
#include <ds.h>

int ds_clear( ds_t dsdes,
              const char* variable_name );
```

Arguments:

dsdes

A data server descriptor returned by [ds_register\(\)](#) (p. 480).

variable_name

The name of the variable that you want to delete.

Library:

libds

Use the -l ds option to qcc to link against this library.

Description:

The *ds_clear()* function deletes *variable_name* from the data server identified by *dsdes*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor *dsdes*.

ESRCH

The variable doesn't exist in the data server.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ds_create()

Create a data server variable

Synopsis:

```
#include <ds.h>

int ds_create( ds_t dsdes,
               const char * variable_name,
               char flags,
               struct sigevent * sigevent );
```

Arguments:

dsdes

A data server descriptor returned by [ds_register\(\)](#) (p. 480).

variable_name

The name of the variable that you want to create.

All variables are global, so only one instance of the variable can exist in the data server process. The maximum length of a variable name is 60 characters.

flags

Flags that specify the variable's behavior:

- **DS_PERM**—don't delete the variable when the application that created it terminates. The variable is removed when the data server process terminates, or if the flag is turned off after the application that created the variable terminates.

If *flags* is 0, the variable is removed if you call [ds_deregister\(\)](#) (p. 474), or the process terminates.

sigevent

A pointer to a [sigevent](#) (p. 2961) structure that describes a proxy or signal to be sent to the external application that created the variable if the data referenced by the variable changes; see below.

Library:

libds

Use the `-l ds` option to `qcc` to link against this library.

Description:

The `ds_create()` function creates a variable, whose name is given by *variable_name*, on the data server identified by *dsdes*.

If the data referenced by *variable_name* changes, a proxy or signal, described in the [sigevent](#) (p. 2961) structure, can be sent to the external application that created *variable_name* (see [ds_set\(\)](#) (p. 481)).

We recommend the following event types for use with this function:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE
- SIGEV_INTR

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid file descriptor *dsdes*.

EEXIST

The variable name already exists in the data server.

ENOMEM

Not enough memory to create the variable or initialize the data.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

ds_deregister()

Deregister an application with the data server

Synopsis:

```
#include <ds.h>

int ds_deregister( ds_t dsdes );
```

Arguments:

dsdes

A data server descriptor returned by [ds_register\(\)](#) (p. 480).

Library:

libds

Use the -l ds option to qcc to link against this library.

Description:

The *ds_deregister()* function deregisters your application with the data server, *dsdes*, and deletes any variables that the data server process created, except those with the DS_PERM flag set (see [ds_flags\(\)](#) (p. 476)).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor, *dsdes*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ds_flags()

Set the flags for a data server variable

Synopsis:

```
#include <ds.h>

int ds_flags( ds_t dsdes,
              const char* variable_name,
              char flags );
```

Arguments:

dsdes

A data server descriptor returned by [ds_register\(\)](#) (p. 480).

variable_name

The name of the data server variable.

flags

The new flags for the variable. The flags include:

- **DS_PERM** — don't delete the variable when the application that created it terminates. The variable is removed when the data server process terminates, or if the flag is turned off after the application that created the variable terminates.

Library:

libds

Use the `-l ds` option to `qcc` to link against this library.

Description:

The *ds_flags()* function changes the state of the *flags* belonging to the variable called *variable_name* on the data server identified by *dsdes*.

Returns:

0 for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor *dsdes*.

ESRCH

The variable doesn't exist in the data server.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ds_get()

Retrieve a data server variable

Synopsis:

```
#include <ds.h>

int ds_get( ds_t dsdes,
            const char* variable_name,
            const char* variable_data,
            size_t data_len );
```

Arguments:

dsdes

A data server descriptor returned by [ds_register\(\)](#) (p. 480).

variable_name

The name of the data server variable that you want to get.

variable_data

A buffer where the function can store the data associated with the variable.

data_len

The size of the buffer, in bytes.

Library:

libds

Use the -l ds option to qcc to link against this library.

Description:

The *ds_get()* function retrieves the data corresponding to *variable_name* from the data server *dsdes*, and places it in the buffer pointed to by *variable_data*.

Returns:

The amount of data written to the buffer *variable_data*, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor *dsdes*.

EMSGSIZE

The buffer isn't big enough for the data.

ESRCH

The variable doesn't exist in the data server.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ds_register()

Register an application with the data server

Synopsis:

```
#include <ds.h>

ds_t ds_register( void );
```

Library:

libds
Use the -l ds option to qcc to link against this library.

Description:

The *ds_register()* function registers your application with the data server. The data server must reside on the same node as your application.

Returns:

A data server descriptor, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

ENOENT

No such file or directory; the data server isn't started.

ENOMEM

Insufficient memory is available to communicate with the data server.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ds_set()

Set a data server variable

Synopsis:

```
#include <ds.h>

int ds_set( ds_t dsdes,
            const char* variable_name,
            const char* variable_data,
            size_t data_len );
```

Arguments:

dsdes

A data server descriptor returned by [ds_register\(\)](#) (p. 480).

variable_name

The name of the data server variable that you want to set.

variable_data

A pointer to the data you want to associate with the variable.

data_len

The size of the data, in bytes.

Library:

libds

Use the -l ds option to qcc to link against this library.

Description:

The *ds_set()* function passes the data *variable_data* to the data server identified by *dsdes*. The data server stores the data in the variable whose name is given by *variable_name*, overwriting any existing value.

Returns:

0 for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor *dsdes*.

ENOMEM

Not enough memory to store the data.

ESRCH

The variable doesn't exist in the data server.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

dup()

Duplicate a file descriptor

Synopsis:

```
#include <unistd.h>

int dup( int filedes );
```

Arguments:

filedes

The file descriptor that you want to duplicate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dup()` function duplicates the file descriptor specified by *filedes*. The new file descriptor refers to the same open file descriptor as the original, and shares any locks. The new file descriptor also:

- references the same file or device
- has the same open mode (read and/or write)
- has an identical file position to the original (changing the position with one descriptor results in a changed position in the other).

Changing the file position with one descriptor results in a changed position for the other.

Calling:

```
dup_filedes = dup( filedes );
```

is the same as:

```
dup_filedes = fcntl( filedes, F_DUPFD, 0 );
```

Returns:

The new file descriptor for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

The file descriptor, *filedes*, isn't a valid.

EMFILE

There are already OPEN_MAX file descriptors in use.

ENOSYS

The *dup()* function isn't implemented for the filesystem specified by *filedes*.

Examples:

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    int filedes, dup_filedes;

    filedes= open( "file",
                  O_WRONLY | O_CREAT | O_TRUNC,
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    if( filedes != -1 ) {
        dup_filedes = dup( filedes );
        if( dup_filedes != -1 ) {
            /* process file */
            /* ... */

            close( dup_filedes );
        }
        close( filedes );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

dup2()

Duplicate a file descriptor, specifying the new descriptor

Synopsis:

```
#include <unistd.h>

int dup2( int filedes,
          int filedes2 );
```

Arguments:

filedes

The file descriptor that you want to duplicate.

filedes2

The number that you want to use for the new file descriptor.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dup2()` function duplicates the file descriptor specified by *filedes*. The number of the new file descriptor will be *filedes2*. If a file already is opened with this descriptor, the file is closed before the duplication is attempted, unless *filedes2* is the same as *filedes*, in which case `dup2()` returns *filedes2* without closing the file.

The new file descriptor:

- references the same file or device
- has the same open mode (read and/or write)
- has an identical file position to the original (changing the position with one descriptor results in a changed position in the other).

Returns:

The value of *filedes2* for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

The *filedes* argument isn't a valid open file descriptor, or *filedes2* is out of range.

EINTR

The function was interrupted by a signal.

Examples:

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    int filedes, dup_filedes;

    filedes = open( "file",
        O_WRONLY | O_CREAT | O_TRUNC,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    if( filedes != -1 ) {
        dup_filedes = 4;
        if( dup2( filedes, dup_filedes ) != -1 ) {
            /* process file */
            /* ... */

            close( dup_filedes );
        }
        close( filedes );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Chapter 7

E

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “E”.

eaccess()

Check to see if a file or directory can be accessed (extended version)

Synopsis:

```
#include <libgen.h>
#include <unistd.h>

int eaccess( const char * path,
             int amode );
```

Arguments:

path

The path to the file or directory that you want to access.

amode

The access mode you want to check. This must be either:

- **F_OK** — test for file existence.

or a bitwise ORing of the following access permissions to be checked, as defined in the header `<unistd.h>`:

- **R_OK** — test for read permission.
- **W_OK** — test for write permission.
- **X_OK** — for a directory, test for search permission. Otherwise, test for execute permission.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `eaccess()` function is an extended version of [access\(\)](#) (p. 61). It checks if the file or directory specified by *path* exists and if it can be accessed with the file access permissions given by *amode*. However, unlike `access()`, it uses the effective user ID and effective group ID.

Returns:

0

The file or directory exists and can be accessed with the specified mode.

-1

An error occurred (*errno* is set.)

Errors:**EACCES**

The permissions specified by *amode* are denied, or search permission is denied on a component of the path prefix.

EINVAL

An invalid value was specified for *amode*.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of the *path* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENOENT

A component of the path isn't valid.

ENOSYS

The *eaccess()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix isn't a directory.

EROFS

Write access was requested for a file residing on a read-only file system.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

_edata

The end of the data segment, excluding BSS data

Synopsis:

N/A

Description:

This linker symbol defines the end of the data segment, excluding BSS data. This variable isn't defined in any header file.

Classification:

[QNX Neutrino](#) (p. 46)

encrypt()

Encrypt or decrypt a string

Synopsis:

```
#include <unistd.h>

void encrypt( char block[64],
              int flag );
```

Arguments:

block

A 64-character array of binary values to encrypt. The function stores the encrypted value in the same array.

flag

0 if you want to encrypt *block*, nonzero to decrypt it.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *encrypt()* function uses the NBS Data Encryption Standard (DES) algorithm and the key you specify by calling [setkey\(\)](#) (p. 2877) to encrypt (if *flags* is zero) or decrypt (if *flags* is nonzero) the given block of data.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

_end

The end of the data segment, including BSS data

Synopsis:

N/A

Description:

This linker symbol defines the end of the data segment, including BSS data. This variable isn't defined in any header file.

Classification:

[QNX Neutrino](#) (p. 46)

endfsent()

Close the filesystem table (/etc/fstab) file

Synopsis:

```
#include <fstab.h>

void endfsent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *endfsent()* routine closes the filesystem table file, `/etc/fstab`.

Classification:

NetBSD (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The functions that work with `/etc/fstab` use static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

endgrent()

Close the group database file

Synopsis:

```
#include <grp.h>

int endgrent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *endgrent()* routine closes the group name database file, so all group access routines behave as if [setgrent\(\)](#) (p. 2862) had never been called.

Returns:

Zero.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

endhostent()

Close the TCP connection and the hosts file

Synopsis:

```
#include <netdb.h>

void endhostent( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *endhostent()* routine closes the TCP connection and the hosts file.

Files:

/etc/hosts

Host database file.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

ENDIAN_BE16()

Return a big-endian 16-bit value in native format

Synopsis:

```
#include <gulliver.h>

uint16_t ENDIAN_BE16( uint16_t num );
```

Arguments:

num

The big-endian number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_BE16()* macro returns the native version of the big-endian value *num*.

Returns:

The native-endian value of *num*.

Examples:

Convert a big-endian value to native-endian:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;

    printf( "0x%04x = 0x%04x\n",
           val, ENDIAN_BE16( val ) );

    return EXIT_SUCCESS;
}
```

On a little-endian system, this prints:

```
0x1234 = 0x3412
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

`ENDIAN_BE16()` is implemented as a macro.

ENDIAN_BE32()

Return a big-endian 32-bit value in native format

Synopsis:

```
#include <gulliver.h>

uint32_t ENDIAN_BE32( uint32_t num );
```

Arguments:

num

The big-endian number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_BE32()* macro returns the native version of the big-endian value *num*.

Returns:

The native-endian value of *num*.

Examples:

Convert a big-endian value to native-endian:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;

    printf( "0x%08x = 0x%08x\n",
           val, ENDIAN_BE32( val ) );

    return EXIT_SUCCESS;
}
```

On a little-endian system, this prints:

```
0xdeadbeef = 0xefbeadde
```


Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_BE32() is implemented as a macro.

ENDIAN_BE64()

Return a big-endian 64-bit value in native format

Synopsis:

```
#include <gulliver.h>

uint64_t ENDIAN_BE64( uint64_t num );
```

Arguments:

num

The big-endian number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_BE64()* macro returns the native version of the big-endian value *num*.

Returns:

The native-endian value of *num*.

Examples:

Convert a big-endian value to native-endian:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678;

    printf( "0x%016llx = 0x%016llx\n",
           val, ENDIAN_BE64( val ) );

    return EXIT_SUCCESS;
}
```

On a little-endian system, this prints:

```
0x1234deadbeef5678 = 0x7856efbeadde3412
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

`ENDIAN_BE64()` is implemented as a macro.

ENDIAN_LE16()

Return a little-endian 16-bit value in native format

Synopsis:

```
#include <gulliver.h>

uint16_t ENDIAN_LE16( uint16_t num );
```

Arguments:

num

The little-endian number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_LE16()* macro returns the native version of the little-endian value *num*.

Returns:

The native-endian value of *num*.

Examples:

Convert a little-endian value to native-endian:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;

    printf( "0x%04x = 0x%04x\n",
           val, ENDIAN_LE16( val ) );

    return EXIT_SUCCESS;
}
```

On a big-endian system, this prints:

```
0x1234 = 0x3412
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

`ENDIAN_LE16()` is implemented as a macro.

ENDIAN_LE32()

Return a little-endian 32-bit value in native format

Synopsis:

```
#include <gulliver.h>

uint32_t ENDIAN_LE32( uint32_t num );
```

Arguments:

num

The little-endian number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_LE32()* macro returns the native version of the little-endian value *num*.

Returns:

The native-endian value of *num*.

Examples:

Convert a little-endian value to native-endian:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;

    printf( "0x%08x = 0x%08x\n",
           val, ENDIAN_LE32( val ) );

    return EXIT_SUCCESS;
}
```

On a big-endian system, this prints:

```
0xdeadbeef = 0xefbeadde
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_LE32() is implemented as a macro.

ENDIAN_LE64()

Return a little-endian 64-bit value in native format

Synopsis:

```
#include <gulliver.h>

uint64_t ENDIAN_LE64( uint64_t num );
```

Arguments:

num

The little-endian number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_LE64()* macro returns the native version of the little-endian value *num*.

Returns:

The native-endian value of *num*.

Examples:

Convert a little-endian value to native-endian:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678;

    printf( "0x%016llx = 0x%016llx\n",
           val, ENDIAN_LE64( val ) );

    return EXIT_SUCCESS;
}
```

On a big-endian system, this prints:

```
0x1234deadbeef5678 = 0x7856efbeadde3412
```


Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_LE64() is implemented as a macro.

ENDIAN_RET16()

Return an endian-swapped 16-bit value

Synopsis:

```
#include <gulliver.h>

uint16_t ENDIAN_RET16( uint16_t num );
```

Arguments:

num

The number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_RET16()* macro returns the endian-swapped value of *num*.

Returns:

The endian-swapped value of *num*.

Examples:

Swap the endianness of a value:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;

    printf( "0x%04x = 0x%04x\n",
           val, ENDIAN_RET16( val ) );

    return EXIT_SUCCESS;
}
```

This prints:

```
0x1234 = 0x3412
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_RET16() is implemented as a macro.

ENDIAN_RET32()

Return an endian-swapped 32-bit value

Synopsis:

```
#include <gulliver.h>

uint32_t ENDIAN_RET32( uint32_t num );
```

Arguments:

num

The number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_RET32()* macro returns the endian-swapped value of *num*.

Returns:

The endian-swapped value of *num*.

Examples:

Swap the endianness of a value:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;

    printf( "0x%08x = 0x%08x\n",
           val, ENDIAN_RET32( val ) );

    return EXIT_SUCCESS;
}
```

This prints:

```
0xdeadbeef = 0xefbeadde
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_RET32() is implemented as a macro.

ENDIAN_RET64()

Return an endian-swapped 64-bit value

Synopsis:

```
#include <gulliver.h>

uint64_t ENDIAN_RET64( uint64_t num );
```

Arguments:

num

The number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ENDIAN_RET64()* macro returns the endian-swapped value of *num*.

Returns:

The endian-swapped value of *num*.

Examples:

Swap the endianness of a value:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678;

    printf( "0x%016llx = 0x%016llx\n",
           val, ENDIAN_RET64( val ) );

    return EXIT_SUCCESS;
}
```

This prints:

```
0x1234deadbeef5678 = 0x7856efbeadde3412
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_RET64() is implemented as a macro.

ENDIAN_SWAP16()

Endian-swap a 16-bit value in place

Synopsis:

```
#include <gulliver.h>

void ENDIAN_SWAP16( uint16_t * num );
```

Arguments:

num

A pointer to the number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ENDIAN_SWAP16()` macro endian-swaps the value pointed to by *num* in place.

Examples:

Swap the endianness of a value:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;
    ENDIAN_SWAP16( &val );

    printf( "val = 0x%04x\n", val );

    return EXIT_SUCCESS;
}
```

This prints:

```
val = 0x3412
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_SWAP16() is implemented as a macro.

ENDIAN_SWAP32()

Endian-swap a 32-bit value in place

Synopsis:

```
#include <gulliver.h>

void ENDIAN_SWAP32( uint32_t * num );
```

Arguments:

num

A pointer to the number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ENDIAN_SWAP32()` macro endian-swaps the value pointed to by *num* in place.

Examples:

Swap the endianness of a value:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;
    ENDIAN_SWAP32( &val );

    printf( "val = 0x%08x\n", val );

    return EXIT_SUCCESS;
}
```

This prints:

```
val = 0xefbeadde
```

Classification:

[*QNX Neutrino*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_SWAP32() is implemented as a macro.

ENDIAN_SWAP64()

Endian-swap a 64-bit value in place

Synopsis:

```
#include <gulliver.h>

void ENDIAN_SWAP64( uint64_t * num );
```

Arguments:

num

A pointer to the number you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ENDIAN_SWAP64()` macro endian-swaps the value pointed to by *num* in place.

Examples:

Swap the endianness of a value:

```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678LL;
    ENDIAN_SWAP16( &val );

    printf( "val = 0x%016x\n", val );

    return EXIT_SUCCESS;
}
```

This prints:

```
val = 0x7856efbeadde3412
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

ENDIAN_SWAP64() is implemented as a macro.

endnetent()

Close the network name database file

Synopsis:

```
#include <netdb.h>

void endnetent( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *endnetent()* routine closes the network name database file.

Files:

/etc/networks

Network name database file.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

endprotoent()

Close the protocol name database file

Synopsis:

```
#include <netdb.h>

void endprotoent( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *endprotoent()* routine closes the protocol name database file.

Files:

/etc/protocols

Protocol name database file.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

endpwent()

Close the password database file

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

int endpwent( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *endpwent()* function closes the password name database file, so all password access routines behave as if [setpwent\(\)](#) (p. 2894) had never been called.

Returns:

Zero.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

endservent()

Close the network services database file

Synopsis:

```
#include <netdb.h>

void endservent( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *endservent()* routine closes the network services database file.

Files:

/etc/services

Network services database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

endspent()

Close the shadow password database file

Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

void endspent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *endspent()* function closes the shadow password database file, so all password access routines behave as if [setspent\(\)](#) (p. 2912) had never been called.

Returns:

Zero.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

endutent()

Close the current user-information file

Synopsis:

```
#include <utmp.h>

void endutent( void );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *endutent()* function closes the currently open file specified in `_PATH_UTMP`.

Files:

`_PATH_UTMP`

The name of the user information file.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

environ

Pointer to the process's environment variables

Synopsis:

```
#include <unistd.h>

extern char ** environ;
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

When a process begins, an array of strings called the *environment* is made available. This array is pointed to by the external variable *environ*. The strings in the array have the form:

variable=value

and are terminated by a NULL pointer.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Caveats:

Don't modify *environ* directly; use [clearenv\(\)](#) (p. 300), [getenv\(\)](#) (p. 838), [putenv\(\)](#) (p. 2454), [searchenv\(\)](#) (p. 2785), [setenv\(\)](#) (p. 2853), and [unsetenv\(\)](#) (p. 3584).

Changes to the environment affect all threads in a multithreaded process.

eof()

Test if the end-of-file has been reached

Synopsis:

```
#include <unistd.h>

int eof( int filedes );
```

Arguments:

filedes

A file descriptor for the file that you want to check.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *eof()* function is a low-level function that determines if the end of the file specified by *filedes* has been reached.

Input operations set the current file position; you can call the *eof()* function to detect the end of the file before more input operations to prevent attempts at reading beyond the end of the file.

Returns:

1

The end-of-file has been reached.

0

The end-of-file hasn't been reached.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The file descriptor, *filedes*, isn't valid.

Examples:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes, len;
    char buffer[100];

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        while( ! eof( filedes ) ) {
            len = read( filedes, buffer, sizeof(buffer) - 1 );
            buffer[ len ] = '\0';
            printf( "%s", buffer );
        }
        close( filedes );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

erand48()

Generate a pseudo-random double in a thread-safe manner

Synopsis:

```
#include <stdlib.h>

double erand48( unsigned short int xsubi[3] );
```

Arguments:

xsubi

An array that comprises the 48 bits of the initial value that you want to use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *erand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative `double` uniformly distributed over the interval `[0.0, 1.0]`. It's a thread-safe version of [drand48\(\)](#) (p. 468).

The *xsubi* array should contain the desired initial value; this makes *erand48()* thread-safe, and lets you start a sequence of random numbers at any known value.

Returns:

A pseudo-random `double`.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

erf()*, *erff()

Compute the error function of a number

Synopsis:

```
#include <math.h>

double erf ( double x );
float erff ( float x );
```

Arguments:

x

The number for which you want to compute the error function.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *erf()* and *erff()* functions compute the following:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

If *x* is large and the result of *erf()* is subtracted from `1.0`, the results aren't very accurate; use [erfc\(\)](#) (p. 534) instead.

This equality is true: $erf(-x) = -erf(x)$

Returns:

The value of the error function, or `NAN` if *x* is `NAN`.



If an error occurs, these functions return `0`, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to `0`, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

erfc()*, *erfcf()*, *erfcl()

Complementary error function

Synopsis:

```
#include <math.h>

double erfc ( double x );
float erfcf ( float x );
long double erfcl( long double x );
```

Arguments:

x

The number for which you want to compute the complementary error function.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *erfc()*, *erfcf()*, and *erfcl()* functions calculate the complementary error function of *x* (i.e. the result of the error function, [erf\(\)](#) (p. 532), subtracted from 1.0). This is useful because the error function isn't very accurate when *x* is large.

The *erf()* function computes:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

This equality is true: $erfc(-x) = 2 - erf(x)$

Returns:

The value of the error function, or NAN if *x* is NAN. For a correct value that would cause an underflow, these functions return 0.0.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

err(), errx()

Display a formatted error message, and then exit

Synopsis:

```
#include <err.h>

void err( int eval,
          const char *fmt, ...);

void errx( int eval,
            const char *fmt, ...);
```

Arguments:

eval

The value to use as the exit code of the process.

fmt

NULL, or a [printf\(\)](#) (p. 2133)-style string used to format the message.

Additional arguments

As required by the format string.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *err()* and *warn()* family of functions display a formatted error message on [stderr](#) (p. 3175):

- The functions without an x in their names display the string associated with the current value of [errno](#) (p. 539).
- Those with a v are “varargs” functions.
- Those with err exit instead of returning.

Function	errno string?	Varargs?	Exits?
<code>err()</code>	Yes	No	Yes
<code>errx()</code>	No	No	Yes
<code>verr()</code> (p. 3612)	Yes	Yes	Yes
<code>verrx()</code> (p. 3612)	No	Yes	Yes
<code>vwarn()</code> (p. 3645)	Yes	Yes	No
<code>vwarnx()</code> (p. 3645)	No	Yes	No
<code>warn()</code> (p. 3669)	Yes	No	No
<code>warnx()</code> (p. 3669)	No	No	No

The `err()` function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the `fmt` argument isn't `NULL`
- the string associated with the current value of `errno`
- a newline character.

The `errx()` function produces a similar message, except that it doesn't include the string associated with `errno`. The message consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, if the `fmt` argument isn't `NULL`
- a newline character.

The `err()` and `errx()` functions don't return, but exit with the value of the argument `eval`.

Examples:

Display the current `errno` information string and exit:

```
if ((p = malloc(size)) == NULL)
    err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
    err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
    errx(1, "too early, wait until %s", start_time_string);
```

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

errno

Global error variable

Synopsis:

```
#include <errno.h>

extern int errno;

char * const sys_errlist[];
int sys_nerr;
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *errno* variable is set to certain error values by many functions whenever an error has occurred. This variable may be implemented as a macro, but you can always examine or set it as if it were a simple integer variable.



- You can't assume that the value of *errno* is valid unless the function that you've called indicates that an error has occurred. The runtime library never resets *errno* to 0.
- The documentation for a function might list special meanings for certain values of *errno*, but this doesn't mean that these are necessarily the only values that the function might set.
- Each thread in a multi-threaded program has its own error value in its thread local storage. No matter which thread you're in, you can simply refer to *errno* — it's defined in such a way that it refers to the correct variable for the thread. For more information, see “[Local storage for private data](#) (p. 3405)” in the documentation for *ThreadCreate()*.

The following variables are also defined in `<errno.h>`:

sys_errlist

An array of error messages corresponding to *errno*.

sys_nerr

The number of entries in the *sys_errlist* array.

The values for *errno* include at least the following. Some are defined by POSIX, and some are additional values.

Name	Value	Meaning	POSIX?
E2BIG	7	Argument list is too long	Yes
EACCES	13	Permission denied	Yes
EADDRINUSE	248	Address is already in use	Yes
EADDRNOTAVAIL	249	Can't assign requested address	Yes
EADV	68	Advertise error	No
EAFNOSUPPORT	247	Address family isn't supported by protocol family	Yes
EAGAIN	11 (same as EWOULDBLOCK)	Resource is temporarily unavailable; try again	Yes
EALREADY	See “ Changes to EALREADY (p. 547),” below	Operation is already in progress	Yes
EBADE	50	Invalid exchange	No
EBADF	9	Bad file descriptor	Yes
EBADFD	81	FD is invalid for this operation	No
EBADFSYS	302	Corrupted filesystem detected	No
EBADMSG	77	Bad message	Yes
EBADR	51	Invalid request descriptor	No
EBADRPC	272	RPC struct is bad	No
EBADRQC	54	Invalid request code	No
EBADSLT	55	Invalid slot	No

Name	Value	Meaning	POSIX?
EBFONT	57	Bad font-file format	No
EBUSY	16	Device or resource is busy	Yes
ECANCELED	47	Operation canceled	Yes
ECHILD	10	No child processes	Yes
ECHRNG	37	Channel number is out of range	No
ECOMM	70	Communication error occurred on send	No
ECONNABORTED	253	Software caused connection to abort	Yes
ECONNREFUSED	261	Connection refused	Yes
ECONNRESET	254	Connection reset by peer	Yes
ECTRLTERM	310	Remap to the controlling terminal	No
EDEADLK	45	Resource deadlock avoided	Yes
EDEADLOCK	56	File locking deadlock	No
EDESTADDRREQ	239	Destination address is required	Yes
EDOM	33	Math argument is out of domain for the function	Yes
EDQUOT	49	Disk quota exceeded	Yes
EENDIAN	313	Endian not supported	No
EEXIST	17	File exists	Yes
EFAULT	14	Bad address	Yes
EFBIG	27	File is too large	Yes

Name	Value	Meaning	POSIX?
EFPOS	110	File positioning error	No
EHOSTDOWN	264	Host is down	No
EHOSTUNREACH	265	Unable to communicate with remote node	Yes
EIDRM	36	Identifier removed	Yes
EILSEQ	88	Illegal byte sequence	Yes
EINPROGRESS	236	Operation now in progress	Yes
EINTR	4	Interrupted function call	Yes
EINVAL	22	Invalid argument	Yes
EIO	5	I/O error	Yes
EISCONN	256	Socket is already connected	Yes
EISDIR	21	Is a directory	Yes
EL2HLT	44	Level 2 halted	No
EL2NSYNC	38	Level 2 not synchronized	No
EL3HLT	39	Level 3 halted	No
EL3RST	40	Level 3 reset	No
ELIBACC	83	Can't access shared library	No
ELIBBAD	84	Accessing a corrupted shared library	No
ELIBEXEC	87	Attempting to exec a shared library	No
ELIBMAX	86	Attempting to link in too many libraries	No

Name	Value	Meaning	POSIX?
ELIBSCN	85	The .lib section in a.out is corrupted	No
ELNRNG	41	Link number is out of range	No
ELOOP	90	Too many levels of symbolic links or prefixes	Yes
EMFILE	24	Too many open files	Yes
EMLINK	31	Too many links	Yes
EMORE	309	More to do, send message again	No
EMSGSIZE	240	Inappropriate message buffer length	Yes
EMULTIHOP	74	Multihop attempted	Yes
ENAMETOOLONG	78	Filename is too long	Yes
ENETDOWN	250	Network is down	Yes
ENETRESET	252	Network dropped connection on reset	Yes
ENETUNREACH	251	Network is unreachable	Yes
ENFILE	23	Too many open files in the system	Yes
ENOANO	53	No anode	No
ENOBUFS	255	No buffer space available	Yes
ENOCSS	43	No CSI structure available	No
ENODATA	61	No data (for no-delay I/O)	Yes
ENODEV	19	No such device	Yes
ENOENT	2	No such file or directory	Yes

Name	Value	Meaning	POSIX?
ENOEXEC	8	Exec format error	Yes
ENOLCK	46	No locks available	Yes
ENOLIC	311	No license available	No
ENOLINK	67	The link has been severed	Yes
ENOMEM	12	Not enough memory	Yes
ENOMSG	35	No message of desired type	Yes
ENONDP	301	Need an NDP (8087...) to run	No
ENONET	64	Machine isn't on the network	No
ENOPKG	65	Package isn't installed	No
ENOPROTOOPT	242	Protocol isn't available	Yes
ENOREMOTE	300	Must be done on local machine	No
ENOSPC	28	No space left on device	Yes
ENOSR	63	Out of streams resources	Yes
ENOSTR	60	Device isn't a stream	Yes
ENOSYS	89	Function isn't implemented	Yes
ENOTBLK	15	A block device is required	No
ENOTCONN	257	Socket isn't connected	Yes
ENOTDIR	20	Not a directory	Yes
ENOTEMPTY	93	Directory isn't empty	Yes

Name	Value	Meaning	POSIX?
ENOTSOCK	238	Socket operation on nonsocket	Yes
ENOTSUP	48	Not supported	Yes
ENOTTY	25	Inappropriate I/O control operation	Yes
ENOTUNIQ	80	Given name isn't unique	No
ENXIO	6	No such device or address	Yes
EOK	0	No error	No
EOPNOTSUPP	103	Operation isn't supported	Yes
EOVERFLOW	79	Value too large to be stored in data type	Yes
EOWNERDEAD	58	The owner of a lock died while holding it	No
EPERM	1	Operation isn't permitted	Yes
EPFNOSUPPORT	246	Protocol family isn't supported	No
EPIPE	32	Broken pipe	Yes
EPROCUNAVAIL	276	Bad procedure for program	No
EPROGMISMATCH	275	Program version wrong	No
EPROGUNAVAIL	274	RPC program isn't available	No
EPROTO	71	Protocol error	Yes
EPROTONOSUPPORT	243	Protocol isn't supported	Yes
EPROTOTYPE	241	Protocol is wrong type for socket	Yes

Name	Value	Meaning	POSIX?
ERANGE	34	Result is too large	Yes
EREMCHG	82	Remote address changed	No
EREMOTE	66	The object is remote	No
ERESTART	91	Restartable system call	No
EROFS	30	Read-only filesystem	Yes
ERPCMISMATCH	273	RPC version is wrong	No
ESHUTDOWN	258	Can't send after socket shutdown	No
ESOCKTNOSUPPORT	244	Socket type isn't supported	No
ESPIPE	29	Illegal seek	Yes
ESRCH	3	No such process	Yes
ESRMNT	69	Server mount error	No
ESRVRFAULT	312	The receive side of a message transfer encountered a memory fault accessing the receive/reply buffer.	No
ESTALE	122	Potentially recoverable I/O error	Yes
ESTRPIPE	92	If pipe/FIFO, don't sleep in stream head	No
ETIME	62	Timer expired	Yes
ETIMEDOUT	260	Connection timed out	Yes
ETOOMANYREFS	259	Too many references: can't splice	No

Name	Value	Meaning	POSIX?
ETXTBSY	26	Text file is busy	Yes
EUNATCH	42	Protocol driver isn't attached	No
EUSERS	94	Too many users (for UFS)	No
EWOULDBLOCK	11 (same as EAGAIN)	Operation would block	Yes
EXDEV	18	Cross-device link	Yes
EXFULL	52	Exchange full	No

Changes to EALREADY

POSIX requires that the error numbers in `<errno.h>` have unique values. In order to satisfy this requirement, we've changed EALREADY so that it no longer has the same value as EBUSY.

This change could cause incompatibility problems in code that's been compiled with the old value of EALREADY. In order to allow applications to safely make the transition to the new value, we've modified `<errno.h>` to define a number of new symbolic values:

EALREADY_OLD

The old value (16).

EALREADY_NEW

The new value (237).

EALREADY_DYNAMIC

A value that you can configure at runtime. This is a variable in `libc` that's set to either the old or new value, based on a system-wide configuration parameter.



In QNX Neutrino 6.4.0, EALREADY was defined as EALREADY_OLD; in QNX Neutrino 6.6.0, it's defined as EALREADY_NEW.

In order to support a safe transition to the new value, you should modify your code as follows:

- If your code checks for EALREADY return codes, recode it to be like this:

```
#if defined(EALREADY_NEW) && defined(EALREADY_OLD)
    if (error == EALREADY_NEW || error == EALREADY_OLD) {
```

```

        // deal with EALREADY error cases
    }
    #else
    if (error == EALREADY) {
        // deal with EALREADY error cases
    }
    #endif

```

This will ensure that the code handles EALREADY error cases from any API that returns either the old or new value.

- If your code returns an EALREADY error, modify it to return EALREADY_DYNAMIC instead. In many cases, you can do this by adding -DEALREADY=EALREADY_DYNAMIC to the compiler flags.

You can use the -e option to procnto to specify the value of EALREADY_DYNAMIC:

-eo

Use the old value, which is the same as that of EBUSY.

-en

Use the POSIX-compliant value.

Examples:

```

/*
 * The following program makes an illegal call
 * to the write() function, then prints the
 * value held in errno.
 */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main( void )
{
    int errvalue;

    errno = EOK;
    write( -1, "hello, world\n",
          strlen( "hello, world\n" ) );
    errvalue = errno;
    printf( "The error generated was %d\n", errvalue );
    printf( "That means: %s\n", strerror( errvalue ) );
}

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

_etext

The end of the text segment

Synopsis:

N/A

Description:

This linker symbol defines the end of the text segment. This variable isn't defined in any header file.

Classification:

[QNX Neutrino](#) (p. 46)

exec()

Execute a file

Synopsis:

```
#include <process.h>

int execl( const char * path,
           const char * arg0,
           const char * arg1,
           ...
           const char * argn,
           NULL );
```

Arguments:

path

The path of the file to execute.

arg0, ..., argn

Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. You must terminate the list with a NULL pointer. The *arg0* argument must point to a filename that's associated with the process being started.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *exec()* function replaces the current process image with a new process image specified by *path*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



In order to start the new process, your process must have the `PROCmgr_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

When a C-language program is executed as a result of this call, it's entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array isn't counted in *argc*.

Multithreaded applications shouldn't use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the *exec** functions are passed on to the new process image in the corresponding *main()* arguments.

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by [close\(\)](#) (p. 336) while locks not affected by *close()* aren't changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren't preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by [atexit\(\)](#) (p. 169) are no longer registered.

If the *path* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *path*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective

group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917)).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see [alarm\(\)](#) (p. 144))
- current working directory
- root directory
- file mode creation mask (see [umask\(\)](#) (p. 3557))
- process signal mask (see [sigprocmask\(\)](#) (p. 3004))
- pending signal (see [sigpending\(\)](#) (p. 3002))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#) (p. 3471))
- resource limits
- controlling terminal
- interval timers

If you call this function from a process with more than one thread, all of the threads are terminated and the new executable image is loaded and executed. No destructor functions are called.

Upon successful completion, the *st_atime* field of the file is marked for update. If the *exec** function failed but was able to locate the process image file, whether the *st_atime* field is marked for update is unspecified. On success, the process image file is considered to be opened with [open\(\)](#) (p. 1905). The corresponding [close\(\)](#) (p. 336) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec** functions.

***exec*()* summary**

Function	Description	POSIX?
<code>exec()</code>	NULL-terminated argument list	Yes
<code>execle()</code> (p. 556)	NULL-terminated argument list, specify the new process environment	Yes
<code>execvp()</code> (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
<code>execvpe()</code> (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process environment	No
<code>execv()</code> (p. 573)	NULL-terminated array of arguments	Yes
<code>execve()</code> (p. 579)	NULL-terminated array of arguments, specify the new process environment	Yes
<code>execvp()</code> (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes
<code>execvpe()</code> (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process environment	No

Returns:

When `exec()` is successful, it doesn't return; otherwise, it returns -1 (`errno` (p. 539) is set).

Errors:**E2BIG**

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *path*, or it doesn't have permission to execute *path*, or *path*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *path* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *path* argument points to an empty string.

ENOEXEC

The new process image file has the correct access permissions, but isn't in the proper format.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Examples:

Replace the current process with **myprog** as if a user had typed:

```
myprog ARG1 ARG2
```

at the shell:

```
#include <stddef.h>
#include <process.h>

execl( "myprog", "myprog", "ARG1", "ARG2", NULL );
```

In this example, myprog will be found if it exists in the current working directory.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

execle()

Execute a file

Synopsis:

```
#include <process.h>

int execle( const char * path,
            const char * arg0,
            const char * arg1,
            ...
            const char * argn,
            NULL,
            const char * envp[] );
```

Arguments:

path

The path of the file to execute.

arg0*, ..., *argn

Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. You must terminate the list with a NULL pointer. The *arg0* argument must point to a filename that's associated with the process being started.

envp

An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the *envp* array with a NULL pointer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *execle()* function replaces the current process image with a new process image specified by *path*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



In order to start the new process, your process must have the PROC_MGR_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

When a C-language program is executed as a result of this call, it's entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array isn't counted in *argc*.

Multithreaded applications shouldn't use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding *main()* arguments.

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by [close\(\)](#) (p. 336) while locks not affected by [close\(\)](#) (p. 336) aren't changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren't preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by [atexit\(\)](#) (p. 169) are no longer registered.

If the *path* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged

for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *path*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917)).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see [alarm\(\)](#) (p. 144))
- current working directory
- root directory
- file mode creation mask (see [umask\(\)](#) (p. 3557))
- process signal mask (see [sigprocmask\(\)](#) (p. 3004))
- pending signal (see [sigpending\(\)](#) (p. 3002))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#) (p. 3471))
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the *st_atime* field of the file is marked for update. If the *exec** function failed but was able to locate the process image file, whether the *st_atime* field is marked for update is unspecified. On success, the process image file is considered to be opened with [open\(\)](#) (p. 1905). The corresponding [close\(\)](#) (p. 336) is

considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec** functions.

exec()* summary

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle()	NULL-terminated argument list, specify the new process's environment	Yes
execp() (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
execpe() (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv() (p. 573)	NULL-terminated array of arguments	Yes
execve() (p. 579)	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp() (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes
execvpe() (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When *execle()* is successful, it doesn't return; otherwise, it returns -1 and sets [errno](#) (p. 539).

Errors:

E2BIG

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *path*, or it doesn't have permission to execute *path*, or *path*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *path* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *path* argument points to an empty string.

ENOEXEC

The new process's image file has the correct access permissions, but isn't in the proper format.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Examples:

Replace the current process with **myprog** as if a user had typed:

```
myprog ARG1 ARG2
```

at the shell:

```
#include <stddef.h>
#include <process.h>

char* env_list[] = { "SOURCE=MYDATA",
                    "TARGET=OUTPUT",
                    "lines=65",
                    NULL
                  };

execle( "myprog",
        "myprog", "ARG1", "ARG2", NULL,
        env_list );
```

In this example, myprog will be found if it exists in the current working directory. The environment for the invoked program consists of the three environment variables **SOURCE**, **TARGET** and **lines**.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

execvp()

Execute a file

Synopsis:

```
#include <process.h>

int execvp( const char * file,
            const char * arg0,
            const char * arg1,
            ...
            const char * argn,
            NULL );
```

Arguments:

file

Used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable ***PATH***.

arg0, ..., argn

Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. Terminate the list terminated with a NULL pointer. The *arg0* argument must point to a filename that's associated with the process.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *execvp()* function replaces the current process image with a new process image specified by *file*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



In order to start the new process, your process must have the PROC_MGR_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

When a C-language program is executed as a result of this call, it's entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array isn't counted in *argc*.

Multithreaded applications shouldn't use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding *main()* arguments.

If the process image file isn't a valid executable object, the contents of the file are passed as standard input to a command interpreter conforming to the [system\(\)](#) (p. 3330) function. In this case, the command interpreter becomes the new process image.

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by [close\(\)](#) (p. 336) while locks not affected by [close\(\)](#) (p. 336) aren't changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren't preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by [atexit\(\)](#) (p. 169) are no longer registered.

If the *file* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *file*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *file*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917)).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see [alarm\(\)](#) (p. 144))
- current working directory
- root directory
- file mode creation mask (see [umask\(\)](#) (p. 3557))
- process signal mask (see [sigprocmask\(\)](#) (p. 3004))
- pending signal (see [sigpending\(\)](#) (p. 3002))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#) (p. 3471))
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the *st_atime* field of the file is marked for update. If the *exec** failed but was able to locate the process image file, whether the *st_atime* field is marked for update is unspecified. On success, the process image file is considered to be opened with *open()* (p. 1905). The corresponding *close()* (p. 336) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec** functions.

exec()* summary

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle() (p. 556)	NULL-terminated argument list, specify the new process's environment	Yes
<i>exec/p()</i>	NULL-terminated argument list, search for the new process in PATH	Yes
exec/pe() (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv() (p. 573)	NULL-terminated array of arguments	Yes
execve() (p. 579)	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp() (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes
execvpe() (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When *exec/p()* is successful, it doesn't return; otherwise, it returns -1 and sets *errno* (p. 539).

Errors:**E2BIG**

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *file*, or it doesn't have permission to execute *file*, or *file*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *file* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *file* argument points to an empty string.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *file* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

execlpe()

Execute a file

Synopsis:

```
#include <process.h>

int execlpe( const char * file,
             const char * arg0,
             const char * arg1,
             ...
             const char * argn,
             NULL,
             const char * envp[] );
```

Arguments:

file

Used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable ***PATH***.

arg0...argn

Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. Terminate the list terminated with a NULL pointer. The *arg0* argument must point to a filename that's associated with the process.

envp

An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the *envp* array with a NULL pointer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

See [exec\(/\)](#) (p. 550) for further information on the `exec*()` family of functions.

The `exec/pe()` function replaces the current process image with a new process image specified by *file*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

- In order to start the new process, your process must have the `PROCmgr_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `exec/pe()` function uses the paths listed in the ***PATH*** environment variable to locate the program to be loaded, provided that the following conditions are met:

- The argument *file* identifies the name of program to be loaded.
- If no path character (/) is included in the name, an attempt is made to load the program from one of the paths in the ***PATH*** environment variable.
- If ***PATH*** isn't defined, the current working directory is used.
- If a path character (/) is included in the name, the program is loaded from the path specified in *file*.

The process is started with the arguments specified in the NULL-terminated arguments *arg1...argn*. *arg0* should point to a filename associated with the program being loaded. Only *arg0* is required, *arg1...argn* are optional.

The new process's environment is specified in *envp*, a NULL-terminated array of NULL-terminated strings. *envp* cannot be NULL, but *envp*[0] can be a NULL pointer if no environment strings are passed.

Each pointer in *envp* points to a string in the form:

variable=value

that is used to define an environment variable.

The environment is the collection of environment variables whose values have been defined with the `export` shell command, the `env` utility, or by the successful execution

of the [putenv\(\)](#) (p. 2454) or [setenv\(\)](#) (p. 2853) functions. A program may read these values with the [getenv\(\)](#) function.

An error is detected when the program cannot be found.

If the *file* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process is set to the owner ID of *file*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *file*. The real user ID, real group ID and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process are saved as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917).

If the calling process had locked any memory, the locks are released.

***exec*()* summary**

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle() (p. 556)	NULL-terminated argument list, specify the new process's environment	Yes
execvp() (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
execpe()	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv() (p. 573)	NULL-terminated array of arguments	Yes
execve() (p. 579)	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp() (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes

Function	Description	POSIX?
execvpe() (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When *exec/pe()* is successful, it doesn't return; otherwise, it returns -1 and sets *errno*.

Errors:**E2BIG**

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *file*, or it doesn't have permission to execute *file*, or *file*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *file* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *file* argument points to an empty string.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *file* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed

because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

execv()

Execute a file

Synopsis:

```
#include <process.h>

int execv( const char * path,
           char * const argv[] );
```

Arguments:

path

A path name that identifies the new process image file.

argv

An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in *argv*[0] must point to a filename that's associated with the process being started.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *execv()* function replaces the current process image with a new process image specified by *path*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



In order to start the new process, your process must have the PROCmgr_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

When a C-language program is executed as a result of this call, it's entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array isn't counted in *argc*.

Multithreaded applications shouldn't use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding *main()* arguments.

The environment for the new process image is taken from the external variable *environ* in the calling process.

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by [close\(\)](#) (p. 336) while locks not affected by [close\(\)](#) (p. 336) aren't changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren't preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by [atexit\(\)](#) (p. 169) are no longer registered.

If the *path* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *path*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective

group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917)).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see [alarm\(\)](#) (p. 144))
- current working directory
- root directory
- file mode creation mask (see [umask\(\)](#) (p. 3557))
- process signal mask (see [sigprocmask\(\)](#) (p. 3004))
- pending signal (see [sigpending\(\)](#) (p. 3002))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#) (p. 3471))
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the *st_atime* field of the file is marked for update. If the *exec** failed but was able to locate the process image file, whether the *st_atime* field is marked for update is unspecified. On success, the process image file is considered to be opened with [open\(\)](#) (p. 1905). The corresponding [close\(\)](#) (p. 336) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec** functions.

***exec*()* summary**

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle() (p. 556)	NULL-terminated argument list, specify the new process's environment	Yes
execvp() (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
execvpe() (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv()	NULL-terminated array of arguments	Yes
execve() (p. 579)	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp() (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes
execvpe() (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When [execv\(\)](#) is successful, it doesn't return; otherwise, it returns -1 and sets [errno](#) (p. 539).

Errors:**E2BIG**

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *path*, or it doesn't have permission to execute *path*, or *path*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *path* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *path* argument points to an empty string.

ENOEXEC

The new process's image file has the correct access permissions, but isn't in the proper format.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Examples:

```
#include <stddef.h>
#include <process.h>

char* arg_list[] = { "myprog", "ARG1", "ARG2", NULL };
execv( "myprog", arg_list );
```

The preceding invokes myprog as if the user entered:

```
myprog ARG1 ARG2
```

as a command at the shell. The program will be found if myprog exists in the current working directory.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

execve()

Execute a file

Synopsis:

```
#include <process.h>

int execve( const char * path,
            char * const argv[],
            char * const envp[] );
```

Arguments:***path***

A path name that identifies the new process image file.

argv

An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in *argv[0]* must point to a filename that's associated with the process being started.

envp

An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the *envp* array with a NULL pointer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *execve()* function replaces the current process image with a new process image specified by *path*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



In order to start the new process, your process must have the `PROC_MGR_AID_SPAWN` ability enabled. For more information, see [proc_mgr_ability\(\)](#) (p. 2141).

When a C-language program is executed as a result of this call, it's entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array isn't counted in *argc*.

Multithreaded applications shouldn't use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding *main()* arguments.

The number of bytes available for the new process's combined argument and environment lists is `ARG_MAX`.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s `FD_CLOEXEC` (p. ?) flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by [close\(\)](#) (p. 336) while locks not affected by [close\(\)](#) (p. 336) aren't changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to `SIG_DFL` in the calling process are set to the default action in the new process image. Signals set to `SIG_IGN` by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren't preserved and the `SA_ONSTACK` flag is cleared for all signals.

After a successful call, any functions previously registered by [atexit\(\)](#) (p. 169) are no longer registered.

If the *path* is on a filesystem mounted with the `ST_NOSUID` flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged

for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *path*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917)).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see [alarm\(\)](#) (p. 144))
- current working directory
- root directory
- file mode creation mask (see [umask\(\)](#) (p. 3557))
- process signal mask (see [sigprocmask\(\)](#) (p. 3004))
- pending signal (see [sigpending\(\)](#) (p. 3002))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#) (p. 3471))
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the *st_atime* field of the file is marked for update. If the *exec** failed but was able to locate the process image file, whether the *st_atime* field is marked for update is unspecified. On success, the process image file is considered to be opened with [open\(\)](#) (p. 1905). The corresponding [close\(\)](#) (p. 336) is considered to

occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec** functions.

exec()* summary

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle() (p. 556)	NULL-terminated argument list, specify the new process's environment	Yes
execvp() (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
execvpe() (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv() (p. 573)	NULL-terminated array of arguments	Yes
execve()	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp() (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes
execvpe() (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When *execve()* is successful, it doesn't return; otherwise, it returns -1 and sets [errno](#) (p. 539).

Errors:

E2BIG

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *path*, or it doesn't have permission to execute *path*, or *path*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *path* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *path* argument points to an empty string.

ENOEXEC

The new process's image file has the correct access permissions, but isn't in the proper format.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

execvp()

Execute a file

Synopsis:

```
#include <process.h>

int execvp( const char * file,
            char * const argv[] );
```

Arguments:

file

Used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable **PATH**.

argv

An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in *argv[0]* must point to a filename that's associated with the process being started.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *execvp()* function replaces the current process image with a new process image specified by *file*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



In order to start the new process, your process must have the PROCmgr_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

When a C-language program is executed as a result of this call, it's entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array isn't counted in *argc*.

Multithreaded applications shouldn't use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding *main()* arguments.

If the process image file isn't a valid executable object, the contents of the file are passed as standard input to a command interpreter conforming to the [system\(\)](#) (p. 3330) function. In this case, the command interpreter becomes the new process image.

The environment for the new process image is taken from the external variable *environ* in the calling process.

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by [close\(\)](#) (p. 336) while locks not affected by [close\(\)](#) (p. 336) aren't changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren't preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by [atexit\(\)](#) (p. 169) are no longer registered.

If the *file* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *file*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *file*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by [setuid\(\)](#) (p. 2917)).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see [alarm\(\)](#) (p. 144))
- current working directory
- root directory
- file mode creation mask (see [umask\(\)](#) (p. 3557))
- process signal mask (see [sigprocmask\(\)](#) (p. 3004))
- pending signal (see [sigpending\(\)](#) (p. 3002))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see [times\(\)](#) (p. 3471))
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the *st_atime* field of the file is marked for update. If the *exec* function failed but was able to locate the process image file, whether the *st_atime* field is marked for update is unspecified. On success, the process image file is

considered to be opened with [open\(\)](#) (p. 1905). The corresponding [close\(\)](#) (p. 336) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec** functions.

exec()* summary

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle() (p. 556)	NULL-terminated argument list, specify the new process's environment	Yes
execvp() (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
execvpe() (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv() (p. 573)	NULL-terminated array of arguments	Yes
execve() (p. 579)	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp()	NULL-terminated array of arguments, search for the new process in PATH	Yes
execvpe() (p. 591)	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When [execvp\(\)](#) is successful, it doesn't return; otherwise, it returns -1 and sets [errno](#) (p. 539).

Errors:

E2BIG

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *file*, or it doesn't have permission to execute *file*, or *file*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *file* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *file* argument points to an empty string.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *file* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

execvpe()

Execute a file

Synopsis:

```
#include <process.h>

int execvpe( const char * file,
             char * const argv[],
             char * const envp[] );
```

Arguments:

file

Used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable **PATH**.

argv

An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in *argv[0]* must point to a filename that's associated with the process being started.

envp

An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the *envp* array with a NULL pointer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:



See [exec\(\)](#) (p. 550) for further information on the `exec*()` family of functions.

The `execvpe()` function replaces the current process image with a new process image specified by *file*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.



- In order to start the new process, your process must have the `PROC_MGR_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `execvpe()` function uses the paths listed in the ***PATH*** environment variable to locate the program to be loaded, provided that the following conditions are met:

- The *file* argument identifies the name of program to be loaded.
- If no path character (/) is included in the name, an attempt is made to load the program from one of the paths in the ***PATH*** environment variable.
- If ***PATH*** isn't defined, the current working directory is used.
- If a path character (/) is included in the name, the program is loaded from the path specified in *file*.

The process is started with the argument specified in *argv*, a NULL-terminated array of NULL-terminated strings. The *argv[0]* entry should point to a filename associated with the program being loaded. The *argv* argument can't be NULL but *argv[0]* can be NULL if no arguments are required.

The new process's environment is specified in *envp*, a NULL-terminated array of NULL-terminated strings. *envp* cannot be NULL, but *envp[0]* can be a NULL pointer if no environment strings are passed.

Each pointer in *envp* points to a string in the form:

variable=value

that's used to define an environment variable.

The environment is the collection of environment variables whose values have been defined with the `export` shell command, the `env` utility, or by the successful execution of the `putenv()` or `setenv()` functions. A program may read these values with the `getenv()` function.

An error is detected if the program can't be found.

If the *file* is on a filesystem mounted with the `ST_NOSUID` flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process is set to the owner ID of *file*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *file*. The real user ID, real group ID and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process are saved as the saved set-user ID and the saved set-group ID used by `setuid()` (p. 2917).

If the calling process had locked any memory, the locks are released.

***exec*()* summary**

Function	Description	POSIX?
exec() (p. 550)	NULL-terminated argument list	Yes
execle() (p. 556)	NULL-terminated argument list, specify the new process's environment	Yes
execvp() (p. 562)	NULL-terminated argument list, search for the new process in PATH	Yes
execvpe() (p. 568)	NULL-terminated argument list, search for the new process in PATH , specify the new process's environment	No
execv() (p. 573)	NULL-terminated array of arguments	Yes
execve() (p. 579)	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp() (p. 585)	NULL-terminated array of arguments, search for the new process in PATH	Yes
execvpe()	NULL-terminated array of arguments, search for the new process in PATH , specify the new process's environment	No

Returns:

When *execvpe()* is successful, it doesn't return; otherwise, it returns -1 and sets *errno*.

Errors:**E2BIG**

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES

The calling process doesn't have permission to search a directory listed in *file*, or it doesn't have permission to execute *file*, or *file*'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *file* or an element of the **PATH** environment variable exceeds PATH_MAX.

ENOENT

One or more components of the pathname don't exist, or the *file* argument points to an empty string.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of *file* isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_exit()

Terminate the program

Synopsis:

```
#include <stdlib.h>

void _exit( int status );
```

Arguments:

status

The exit status to use for the program. The value may be zero, EXIT_STATUS, EXIT_FAILURE or any other value. Note that only the least significant bits (i.e. *status* and 0377) may be available to a waiting parent process.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The `_exit()` function causes normal program termination to occur.



The functions registered with [atexit\(\)](#) (p. 169) aren't called when you use `_exit()` to terminate a program. If you want those functions to be called, use [exit\(\)](#) (p. 599) instead.

The `_exit()` function does the following when a process terminates for any reason:

1. Closes all open file descriptors and directory streams in the calling process.
2. Notifies the parent process of the calling process if the parent called [wait\(\)](#) (p. 3652) or [waitpid\(\)](#) (p. 3666). The low-order 8 bits of *status* are made available to the parent via `wait()` or `waitpid()`.
3. Saves the exit *status* if the parent process of the calling process isn't executing a `wait()` or `waitpid()` function. If the parent calls `wait()` or `waitpid()` later, this status is returned immediately.

4. Sends a **SIGHUP** signal to the calling process's children; this can indirectly cause the children to exit if they don't handle **SIGHUP**. Children of a terminated process are assigned a new parent process.
5. Sends a **SIGCHLD** signal to the parent process.
6. Sends a **SIGHUP** signal to each process in the foreground process group if the calling process is the controlling process for the controlling terminal of that process group.
7. Disassociates the controlling terminal from the calling process's session if the process is a controlling process, allowing it to be acquired by a new controlling process.
8. If the process exiting causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a **SIGHUP** signal followed by a **SIGCONT** signal is sent to each process in the newly-orphaned process group.

Returns:

The `_exit()` function doesn't return.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    FILE *fp;

    if( argc <= 1 ) {
        fprintf( stderr, "Missing argument\n" );
        exit( EXIT_FAILURE );
    }

    fp = fopen( argv[1], "r" );
    if( fp == NULL ) {
        fprintf( stderr, "Unable to open '%s'\n", argv[1] );
        _exit( EXIT_FAILURE );
    }
    fclose( fp );

    /*
     * At this point, calling _exit() is the same as calling
     * return EXIT_SUCCESS;...
     */
    _exit( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

exit()

Exit the calling program

Synopsis:

```
#include <stdlib.h>

void exit( int status );
```

Arguments:

status

The exit status to use for the program. The value may be zero, EXIT_STATUS, EXIT_FAILURE or any other value. Note that only the least significant bits (i.e. *status* and 0377) may be available to a waiting parent process.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *exit()* function causes the calling program to exit normally. When a program exits normally:

1. All functions registered with the [atexit\(\)](#) (p. 169) function are called.
2. All open file streams (those opened by [fopen\(\)](#) (p. 681), [fdopen\(\)](#) (p. 635), [freopen\(\)](#) (p. 721), or [popen\(\)](#) (p. 2019)) are flushed and closed.
3. All temporary files created by the [tmpfile\(\)](#) (p. 3478) function are removed.
4. The return *status* is made available to the parent process; *status* is typically set to EXIT_SUCCESS to indicate successful termination and set to EXIT_FAILURE or some other value to indicate an error.

Returns:

The *exit()* function doesn't return.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main( int argc, char *argv[] )
{
    FILE *fp;

    if( argc <= 1 ) {
        fprintf( stderr, "Missing argument\n" );
        exit( EXIT_FAILURE );
    }

    fp = fopen( argv[1], "r" );
    if( fp == NULL ) {
        fprintf( stderr, "Unable to open '%s'\n", argv[1] );
        exit( EXIT_FAILURE );
    }
    fclose( fp );
    exit( EXIT_SUCCESS );

    /*
    You'll never get here; this prevents compiler
    warnings about "function has no return value".
    */
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

exp(), expf(), expl()

Compute the exponential function of a number

Synopsis:

```
#include <math.h>

double exp( double x );
float expf( float x );
long double expl( long double x );
```

Arguments:

x

The number for which you want to calculate the exponential.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the exponential function of x (e^x).

A range error occurs if the magnitude of x is too large.

Returns:

The exponential value of x . For a correct value that would cause an underflow, these functions return 0.0.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
```

```
    printf( "%f\n", exp(.5) );  
    return EXIT_SUCCESS;  
}
```

produces the output:

1.648721

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The value of $\expm1(x)$ may be more accurate than $\exp(x) - 1.0$ for small values of x .

expm1(), expm1f()

Compute the exponential of a number, then subtract 1

Synopsis:

```
#include <math.h>

double expm1 ( double x );
float expm1f ( float x );
```

Arguments:

x

The number for which you want to calculate the exponential minus one.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *expm1()* and *expm1f()* functions compute the exponential of *x*, minus 1 ($e^x - 1$).

A range error occurs if the magnitude of *x* is too large.

The value of *expm1(x)* may be more accurate than *exp(x) - 1.0* for small values of *x*.

The *expm1()* and [log1p\(\)](#) (p. 1543) functions are useful for financial calculations of $((1+x)^n - 1)/x$, namely:

```
expm1(n * log1p(x))/x
```

when *x* is very small (for example, when performing calculations with a small daily interest rate). These functions also simplify writing accurate inverse hyperbolic functions.

Returns:

The exponential value of *x*, minus 1.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;

    a = 2;
    b = expm1(a);
    printf("(e ^ %f) -1 is %f \n", a, b);

    return(0);
}
```

produces the output:

```
(e ^ 2.000000) -1 is 6.389056
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 8

F

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>FD_SET()</i>	select() (p. 2791)
<i>FD_CLR()</i>	select() (p. 2791)
<i>FD_ISSET()</i>	select() (p. 2791)
<i>FD_ZERO()</i>	select() (p. 2791)
<i>fstab</i>	getfsent() (p. 844)

The following have been deprecated:

Instead of:	Use:
<i>finite()</i> , <i>finitef()</i>	isfinite() (p. 1437)

This chapter describes the API components whose names begin with “F”.

fabs(), fabsf()

Compute the absolute value of a double number

Synopsis:

```
#include <math.h>

double fabs( double x );
float fabsf( float x );
```

Arguments:

x

The number you want the absolute value of.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the absolute value of *x*.

Returns:

The absolute value of *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f %f\n", fabs(.5), fabs(-.5) );
    return EXIT_SUCCESS;
}
```

produces the output:

0.500000 0.500000

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fcfgopen()

Open a configuration file

Synopsis:

```
#include <cfgopen.h>

FILE * fcfgopen( const char * path,
                 const char * mode,
                 int location,
                 const char * historical,
                 char * namebuf,
                 int nblen );
```

Arguments:

path

The name of the configuration file that you want to open.

mode

A string that describes the mode to open in; see [fopen\(\)](#) (p. 681).

location

Flags that describe how the path is constructed. See “[Search condition flags](#) (p. 268)” in the documentation for *cfgopen()*.

historical

A optional file to open as a last resort if none of the criteria for finding the path is met. This string works like a path search order, and lets you search more than one location. You can also specify %H to substitute the hostname value into the string. Specify NULL to ignore this option.

namebuf

A buffer to save the pathname in. Specify NULL to ignore this option.

nblen

The length of the buffer pointed to by *namebuf*. Specify 0 to ignore this option.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `fcfgopen()` function is similar to `cfgopen()` (p. 266) with these exceptions:

- The `CFGFILE_NOFD` flag isn't valid.
- The values for *flags* described in `open()` (p. 1905) aren't valid.

Returns:

A valid *fd* if `CFGFILE_NOFD` isn't specified, a nonnegative value if `CFGFILE_NOFD` is specified, or -1 if an error occurs.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fchdir()

Change the working directory

Synopsis:

```
#include <unistd.h>

int fchdir(int fd);
```

Arguments:

var

A file descriptor for the directory that you want to become the current working directory.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *fchdir()* function is similar to [chdir\(\)](#) (p. 286), except that you use a file descriptor to specify the directory that you want to become the new current working directory. You can obtain a file descriptor for a directory by calling [open\(\)](#) (p. 1905), provided that the file status flags and access modes don't contain `O_WRONLY` or `O_RDWR`.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set). If the function fails, the current working directory isn't changed.

Errors:

EACCES

Search permission is denied for the directory referenced by *fd*.

EBADF

The *fd* argument isn't an open file descriptor.

ENOTDIR

The open file descriptor *fd* doesn't refer to a directory.

EINTR

A signal was caught during the execution of *fchdir()*.

EIO

An I/O error occurred while reading from or writing to the filesystem.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fchmod()

Change the permissions for a file

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>

int fchmod( int fd,
            mode_t mode );
```

Arguments:

fd

A file descriptor for the file whose permissions you want to change.

mode

The new permissions for the file. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fchmod()* function changes the permissions for a file referred to by *fd* to be the settings in the mode given by *mode*.

If the effective user ID of the calling process is equal to the file owner, or the calling process has appropriate privileges (for example, the superuser), *fchmod()* sets the `S_ISUID`, `S_ISGID` and the file permission bits, defined in the `<sys/stat.h>` header file, from the corresponding bits in the *mode* argument. These bits define access permissions for the user associated with the file, the group associated with the file, and all others.

For a regular file, if the calling process doesn't have appropriate privileges, and if the group ID of the file doesn't match the effective group ID, the `S_ISGID` (set-group-ID on execution) bit in the file's mode is cleared upon successful return from *fchmod()*.

Changing the permissions has no any effect on any file descriptors for files that are already open.

If *fchmod()* succeeds, the *st_ctime* field of the file is marked for update.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EBADF

Invalid file descriptor.

ENOSYS

The *fchmod()* function isn't implemented for the filesystem specified by *fd*.

EPERM

The effective user ID doesn't match the owner of the file, and the calling process doesn't have appropriate privileges.

EROFS

The referenced file resides on a read-only filesystem.

Examples:

```
/*
 * Change the permissions of a list of files
 * to be read/write by the owner only
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;
    int fd;

    for( i = 1; i < argc; i++ ) {
        if( ( fd = open( argv[i], O_RDONLY ) ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
        else if( fchmod( fd, S_IRUSR | S_IWUSR ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
        close( fd );
    }
    return ecode;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fchown()

Change the user ID and group ID of a file

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

int fchown( int fd,
            uid_t owner,
            gid_t group );
```

Arguments:

fd

A file descriptor for the file whose ownership you want to change.

owner

The user ID of the new owner.

group

The group ID of the new owner.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *fchown()* function changes the user ID and group ID of the file referenced by *fd* to be the numeric values contained in *owner* and *group*, respectively.

Only processes with an effective user ID equal to the user ID of the file, or with appropriate privileges (for example, the superuser) may change the ownership of a file.

The `_POSIX_CHOWN_RESTRICTED` flag is enforced. This means that only the superuser may change the ownership of a file. The group of a file may be changed by the superuser, or also by a process with the effective user ID equal to the user ID of the file, if (and only if) *owner* is equal to the user ID of the file and *group* is equal to the effective group ID of the calling process.

If the *fd* argument refers to a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode are cleared if the function is successful.

If *fchown()* succeeds, the *st_ctime* field of the file is marked for update.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EBADF

Invalid file descriptor.

EPERM

The effective user ID doesn't match the owner of the file, or the calling process doesn't have appropriate privileges.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
/*
 * Change the ownership of a list of files
 * to the current user/group
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;
    int fd;

    for( i = 1; i < argc; i++ ) {
        if( ( fd = open( argv[i], O_RDONLY ) ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
        else if( fchown( fd, getuid(), getgid() ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }
    close( fd );
    return ecode;
}
```

Classification:*POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fclose()

Close a stream

Synopsis:

```
#include <stdio.h>

int fclose( FILE* fp );
```

Arguments:

fp

The stream you want to close.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fclose()* function closes the stream specified by *fp*. Any unwritten, buffered data is flushed before the file is closed. Any unread, buffered data is discarded.

If the associated buffer was automatically allocated, it's deallocated.

Returns:

0 for success, or EOF if an error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor underlying *fp*, and the process would be delayed in the write operation.

EBADF

The file descriptor underlying *fp* isn't valid.

EFBIG

One of the following:

- An attempt was made to write a file that exceeds the maximum file size.
- An attempt was made to write a file that exceeds the process's file size limit.
- The file is a regular file, and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EINTR

The *fclose()* function was interrupted by a signal.

EIO

One of the following:

- The process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU**, and the process group of the process is orphaned.
- (QNX Neutrino extension) An I/O error occurred when writing cached data to the underlying media.
- (QNX Neutrino extension) An I/O error occurred when updating metadata on the underlying media.
- (QNX Neutrino extension) The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOSPC

There was no free space remaining on the device containing the file.

ENXIO

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EPIPE

An attempt was made to write to a pipe or FIFO that wasn't open for reading by any process. A **SIGPIPE** signal is also sent to the thread.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;

    fp = fopen( "stdio.h", "r" );
    if( fp != NULL ) {
        fclose( fp );
    }

    return EXIT_SUCCESS;
}
```

```
    }  
    return EXIT_FAILURE;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fcloseall()*Close all open stream files***Synopsis:**

```
#include <stdio.h>

int fcloseall( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *fcloseall()* function closes all open streams, except *stdin*, *stdout* and *stderr*. This includes streams created (and not yet closed) by [fdopen\(\)](#) (p. 635), [fopen\(\)](#) (p. 681) and [freopen\(\)](#) (p. 721).

Returns:

0

Errors:

If an error occurs, [errno](#) (p. 539) is set.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "The number of files closed is %d\n", fcloseall() );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

fcntl()

Provide control over an open file

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl( int filde,
           int cmd,
           ... );
```

Arguments:

filde

The descriptor for the file you want to control.

cmd

The command to execute; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fcntl()` function provides control over the open file referenced by file descriptor *filde*. To establish a lock with this function, open with write-only permission (`O_WRONLY` (p. ?)) or with read/write permission (`O_RDWR` (p. ?)).

The type of control is specified by the *cmd* argument, which may require a third data argument (*arg*). The *cmd* argument is defined in `<fcntl.h>`, and includes at least the following values:

F_ALLOCSP

Allocate storage space for the section of the file specified by the *l_start*, *l_len*, and *l_whence* fields of a `struct flock` structure pointed to by the additional argument to `fcntl()`

F_ALLOCSP64

Same as `F_ALLOCSP`, except that the argument is a pointer to a `struct flock64` structure, where the `l_start` and `l_len` fields are 64-bit values.

F_DUPFD

Allocate and return a new file descriptor that's the lowest numbered available (i.e. not already open) file descriptor greater than or equal to the third argument, *arg*, taken as an `int`. The new file descriptor refers to the same file as *fildev*, and shares any locks.

F_FREESP

Free storage space for the section of the file specified by the `l_start`, `l_len`, and `l_whence` fields of the `struct flock` structure pointed to by the additional argument to *fcntl()*.

F_FREESP64

Same as `F_FREESP`, except that the argument is a pointer to a `struct flock64` structure, where the `l_start` and `l_len` fields are 64-bit values.

F_GETFD

Get the file descriptor flags associated with the file descriptor *fildev*. File descriptor flags are associated with a single file descriptor, and don't affect other file descriptors referring to the same file.

F_GETFL

Get the file status flags and the file access modes associated with *fildev*. The flags and modes are defined in `<fcntl.h>`.

The file status flags (see [open\(\)](#) (p. 1905) for more detailed information) are:

- `O_APPEND` (p. ?) — Set append mode.
- `O_NONBLOCK` (p. ?) — No delay.

The file access modes are:

- `O_RDONLY` (p. ?) — Open for reading only.
- `O_RDWR` (p. ?) — Open for reading and writing.
- `O_WRONLY` (p. ?) — Open for writing only.

F_GETLK

Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type `struct flock` (defined in `<fcntl.h>`). For more information, see the [flock structure](#) (p. 626) section below. The information returned overwrites the information passed to *fcntl()* in the structure pointed to by *arg*.

If no lock is found that prevents this lock from being created, the structure is left unchanged, except for the lock type, which is set to **F_UNLCK** (p. ?). If a lock is found, the *l_pid* member of the structure pointed to by *arg* is set to the process ID of the process holding the blocking lock and *l_whence* is set to **SEEK_SET** (p. ?).

F_GETLK64

Same as **F_GETLK**, except that the argument is a pointer to a **struct flock64** structure, where the *l_start* and *l_len* fields are 64-bit values.

F_SETFD

Set the file descriptor flags associated with *fildev* to the third argument, *arg*, taken as type **int**. See the above discussion for more details.

The only defined file descriptor flag is:

FD_CLOEXEC

When this flag is clear, the file remains open across *spawn*()* or *exec*()* calls; else the file is closed.

F_SETFL

Set the file status flags, as shown above, for the open file description associated with *fildev* from the corresponding bits in the third argument, *arg*, taken as type **int**. You can't use this function to change the file access mode. All bits set in *arg*, other than the file status bits, are ignored.

F_SETLK

Set or clear a file segment lock, according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, as defined in the header file **<fcntl.h>**, and documented below. This command is used to create the following locks (defined in **<fcntl.h>**):

F_RDLCK

Shared or read locks.

F_UNLCK

Remove either type of lock.

F_WRLCK

Exclusive or write locks.

If a lock can't be set, *fcntl()* returns immediately.

F_SETLK64

Same as F_SETLK, except that the argument is a pointer to a `struct flock64` structure, where the `l_start` and `l_len` fields are 64-bit values.

F_SETLKW

This command is the same as [F_SETLK](#) (p. ?), except that when a lock is blocked by other locks, the process waits until the request can be satisfied. If a signal that's to be caught is received while `fcntl()` is waiting for a region, the call is interrupted without performing the lock operation, and `fcntl()` returns -1 with `errno` set to EINTR.

F_SETLKW64

Same as F_GETLKW, except that the argument is a pointer to a `struct flock64` structure, where the `l_start` and `l_len` fields are 64-bit values.

flock structure

The `flock` structure contains at least the following members:

short l_type

One of [F_RDLCK](#) (p. ?), [F_WRLCK](#) (p. ?) or [F_UNLCK](#) (p. ?).

short l_whence

One of the following flags that specify where the relative offset, `l_start`, is measured from:

SEEK_CUR

Current seek position.

SEEK_END

End of file.

SEEK_SET

Start of file.

off_t l_start

Relative offset in bytes.

off_t l_len

Consecutive bytes to lock; if 0, then until EOF; if negative, the preceding bytes up to, but not including, the start byte.

pid_t l_pid

Process ID of the process holding the lock, returned when *cmd* is [F_GETLK](#) (p. ?).

When a shared lock is set on a segment of a file, other processes can set shared locks on the same segment, or a portion of it. A shared lock prevents other processes from setting exclusive locks on any portion of the protected area. A request for a shared lock fails if the file was opened write-only.

An exclusive lock prevents any other process from setting a shared or an exclusive lock on a portion of the protected area. A request for an exclusive lock fails if the file was opened read-only.

Locks may start and extend beyond the current end of file, but may not start or extend before the beginning of the file; to attempt to do so is an error. A lock extends to “infinity” (the largest possible value for the file offset) if *l_len* is set to zero. If *l_whence* and *l_start* point to the beginning of the file, and *l_len* is zero, the entire file is locked.

The calling process may have only one type of lock set for each byte of a file. Before successfully returning from an [F_SETLK](#) (p. ?) or [F_SETLKW](#) (p. ?) request, the previous lock type (if any) for each byte in the specified lock region is replaced by the new lock type. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by the process, or the process holding the file descriptor terminates. Locks aren't inherited by a child process using the *fork()* function. However, locks are inherited across *exec*()* or *spawn*()* calls.



A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process's locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, *fcntl()* fails with EDEADLK. However, the system can't always detect deadlocks in the network case, and care should be exercised in the design of your application for this possibility.



Locking is a protocol designed for updating a file shared among concurrently running applications. Locks are only advisory, that is, they don't prevent an errant or poorly-designed application from overwriting a locked region of a shared file. An application should use locks to indicate regions of a file that are to be updated by the application, and it should respect the locks of other applications.

The following functions ignore locks:

- [chsize\(\)](#) (p. 298)
- [ltrunc\(\)](#) (p. 1562)
- [open\(\)](#) (p. 1905)

- [read\(\)](#) (p. 2502)
 - [sopen\(\)](#) (p. 3083)
 - [write\(\)](#) (p. 3737)
-

Returns:

-1 if an error occurred ([errno](#) (p. 539) is set). The successful return value(s) depend on the request type specified by *arg*, as shown in the following table:

[F_DUPFD](#) (p. ?)

A new file descriptor.

[F_GETFD](#) (p. ?)

Value of the file descriptor flags (never a negative value).

[F_GETFL](#) (p. ?)

Value of the file status flags and access modes as shown above (never a negative value).

[F_GETLK](#) (p. ?)

Value other than -1.

[F_SETFD](#) (p. ?)

Value other than -1.

[F_SETFL](#) (p. ?)

Value other than -1.

[F_SETLK](#) (p. ?)

Value other than -1.

[F_SETLKW](#) (p. ?)

Value other than -1.

Errors:**EAGAIN**

The *cmd* argument is [F_SETLK](#) (p. ?), the type of lock (*l_type*) is a shared lock ([F_RDLCK](#) (p. ?)), and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and

some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

EBADF

One of the following occurred:

- The *fildest* argument isn't a valid file descriptor.
- The *cmd* argument is `F_SETLK` (p. ?) or `F_SETLKW` (p. ?), the type of lock (*l_type*) is a shared lock (`F_RDLCK` (p. ?)), and *fildest* isn't a valid file descriptor open for reading.
- The *cmd* argument is `F_SETLK` (p. ?) or `F_SETLKW` (p. ?), the type of lock (*l_type*) is an exclusive lock (`F_WRLCK` (p. ?)), and *fildest* isn't a valid file descriptor open for writing.

EDEADLK

The *cmd* argument is `F_SETLKW` (p. ?), and a deadlock condition was detected.

EINTR

The *cmd* argument is `F_SETLKW` (p. ?), and the function was interrupted by a signal.

EINVAL

One of the following occurred:

- The *cmd* argument is invalid.
- The *cmd* argument is `F_DUPFD` (p. ?), and the third argument is negative, or greater than the configured number of maximum open file descriptors per process.
- The *cmd* argument is `F_GETLK` (p. ?), `F_SETLK` (p. ?) or `F_SETLKW` (p. ?), and the data *arg* isn't valid, or *fildest* refers to a file that doesn't support locking.

EMFILE

The *cmd* argument is `F_DUPFD` (p. ?), and the process has no unused file descriptors, or no file descriptors greater than or equal to *arg* are available.

ENOLCK

The *cmd* argument is `F_SETLK` (p. ?) or `F_SETLKW` (p. ?), and satisfying the lock or unlock request causes the number of lock regions in the system to exceed the system-imposed limit.

ENOSYS

The filesystem doesn't support the operation.

EOverflow

One of the values to be returned can't be represented correctly.

Examples:

```
/*
 * This program makes "stdout" synchronous
 * to guarantee the data is recoverable
 * (if it's redirected to a file).
 */
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int flags, retval;

    flags = fcntl( STDOUT_FILENO, F_GETFL );

    flags |= O_DSYNC;

    retval = fcntl( STDOUT_FILENO, F_SETFL, flags );
    if( retval == -1 ) {
        printf( "error setting stdout flags\n" );
        return EXIT_FAILURE;
    }

    printf( "Hello, QNX Neutrino world.\n" );

    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

A cancellation point occurs when a thread is executing *fcntl()* if the *cmd* argument is *F_SETLKW*; a cancellation point may occur in the case of [F_DUPFD](#) (p. ?) (when dupping across the network), [F_GETFD](#) (p. ?), and [F_SETFD](#) (p. ?).

fdatasync()

Synchronize file data

Synopsis:

```
#include <unistd.h>

int fdatasync( int filedes );
```

Arguments:

filedes

The descriptor of the file that you want to synchronize.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fdatasync()* function forces all queued I/O operations for the file specified by the *filedes* file descriptor to finish, synchronizing the file's data. The function blocks until this is finished. For more information about synchronizing, see “Filesystems and block I/O (devb-*) drivers” in the Fine-Tuning Your System chapter of the QNX Neutrino *User's Guide*.

This function is similar to [fsync\(\)](#) (p. 787), except that *fsync()* also guarantees the integrity of file information, such as access and modification times.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The specified *filedes* isn't a valid file descriptor open for writing.

EINVAL

The implementation doesn't support synchronized I/O for the given file.

ENOSYS

The *fdatsync()* function isn't supported for the filesystem specified by *filedes*.

Classification:

[*POSIX 1003.1 SIO*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fdistrusted()

Determine if a file is trusted

Synopsis:

```
#include <unistd.h>

int fdistrusted( int fd );
```

Arguments:

fd

A descriptor for the file that you want to check.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fdistrusted()* function determines whether or not the file associated with the given file descriptor is trusted or is in a trusted filesystem.

If a process with any privileged abilities enabled—see [procmgr_ability\(\)](#) (p. 2141)—attempts to mark a region of memory as `PROT_EXEC`, any memory-mapped files in the region must be trusted or be from a trusted filesystem.

You can use the `pathtrust` to mark files or filesystems as trusted. If no files or filesystems are designated as trusted, all are trusted.

Returns:

EOK if the file is trusted, EPERM if it isn't, or some other [errno](#) (p. 539) value if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fdopen()

Associate a stream with a file descriptor

Synopsis:

```
#include <stdio.h>

FILE* fdopen( int filedes,
              const char* mode );
```

Arguments:

filedes

The file descriptor that you want to associate with a stream.

mode

The mode specified when *filedes* was originally opened. For information, see [fopen\(\)](#) (p. 681), except modes beginning with w don't cause truncation of the file.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *fdopen()* function associates a stream with the file descriptor *filedes*, which represents an opened file or device.

The *filedes* argument is a file descriptor that was returned by one of [accept\(\)](#) (p. 58), [creat\(\)](#) (p. 381), [dup\(\)](#) (p. 483), [dup2\(\)](#) (p. 485), [fcntl\(\)](#) (p. 623), [open\(\)](#) (p. 1905), [pipe\(\)](#) (p. 2011), or [sopen\(\)](#) (p. 3083).

The *fdopen()* function preserves the offset maximum previously set for the open file description corresponding to *filedes*.

Returns:

A file stream for success, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

The *filedes* argument isn't a valid file descriptor.

EINVAL

The *mode* argument isn't a valid mode.

EMFILE

Too many file descriptors are currently in use by this process.

ENOMEM

There isn't enough memory for the FILE structure.

Examples:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes;
    FILE *fp;

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        fp = fdopen( filedes, "r" );
        if( fp != NULL ) {
            /* Also closes the underlying FD, filedes. */
            fclose( fp );
        }
    }
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

feof()

Test a stream's end-of-file flag

Synopsis:

```
#include <stdio.h>

int feof( FILE* fp );
```

Arguments:

fp

The stream you want to test.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `feof()` function tests the end-of-file flag for the stream specified by *fp*.

Because the end-of-file flag is set when an input operation attempts to read past the end-of-file, the `feof()` function detects the end-of-file only after an attempt is made to read beyond the end-of-file. Thus, if a file contains 10 lines, the `feof()` won't detect the end-of-file after the tenth line is read; it will detect the end-of-file on the next read operation.

Returns:

0 if the end-of-file flag isn't set, or nonzero if the end-of-file flag is set.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

void process_record( char *buf )
{
    printf( "%s\n", buf );
}

int main( void )
{
    FILE *fp;
    char buffer[100];

    fp = fopen( "file", "r" );
```

```
fgets( buffer, sizeof( buffer ), fp );
while( ! feof( fp ) ) {
    process_record( buffer );
    fgets( buffer, sizeof( buffer ), fp );
}
fclose( fp );

return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ferror()

Test a stream's error flag

Synopsis:

```
#include <stdio.h>

int ferror( FILE* fp );
```

Arguments:

fp

The stream whose error flag you want to test.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ferror()* function tests the error flag for the stream specified by *fp*.

Returns:

0 if the error flag isn't set, or nonzero if the error flag is set.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        c = fgetc( fp );
        if( ferror( fp ) ) {
            printf( "Error reading file\n" );
        }
    }
    fclose( fp );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fflush()

Flush the buffers for a stream

Synopsis:

```
#include <stdio.h>

int fflush( FILE* fp );
```

Arguments:

fp

NULL, or the stream whose buffers you want to flush.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

If the stream specified by *fp* is open for output or update, the *fflush()* function causes any buffered (see [setvbuf\(\)](#) (p. 2920)) but unwritten data to be written to the file descriptor associated with the stream (see [fileno\(\)](#) (p. 661)).

If the file specified by *fp* is open for input or update, the *fflush()* function undoes the effect of any preceding `ungetc` operation on the stream.

If *fp* is NULL, all open streams are flushed.

Returns:

0 for success, or EOF if an error occurs ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor underlying *fp*, and the process would be delayed in the write operation.

EBADF

The file descriptor underlying *fp* isn't valid.

EFBIG

One of the following:

- An attempt was made to write a file that exceeds the maximum file size.
- An attempt was made to write a file that exceeds the process's file size limit.
- The file is a regular file, and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EINTR

The *fflush()* function was interrupted by a signal.

EIO

One of the following:

- The process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU**, and the process group of the process is orphaned.
- (QNX Neutrino extension) An I/O error occurred when writing cached data to the underlying media.
- (QNX Neutrino extension) An I/O error occurred when updating metadata on the underlying media.
- (QNX Neutrino extension) The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOSPC

There was no free space remaining on the device containing the file.

ENXIO

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EPIPE

An attempt was made to write to a pipe or FIFO that wasn't open for reading by any process. A **SIGPIPE** signal is also sent to the thread.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "Press Enter to continue..." );
```

```
    fflush( stdout );  
    getchar();  
    return EXIT_SUCCESS;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

ffs()

Find the first bit set in a bit string

Synopsis:

```
#include <strings.h>
int ffs( int value );
```

Arguments:

value
The bit string.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *ffs()* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting from 1, starting at the rightmost bit.

Returns:

The index of the first bit set, or 0 if *value* is zero.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fgetc()

Read a character from a stream

Synopsis:

```
#include <stdio.h>

int fgetc( FILE* fp );
```

Arguments:

fp

The stream from which you want to read a character.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fgetc()* function reads the next character from the stream specified by *fp*.

Returns:

The next character from *fp*, cast as `(int)(unsigned char)`, or EOF if end-of-file has been reached or if an error occurs (*errno* (p. 539) is set).



Use *feof()* (p. 637) or *ferror()* (p. 639) to distinguish an end-of-file condition from an error.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor underlying *fp*, and the process would be delayed in the write operation.

EBADF

The file descriptor underlying *fp* isn't a valid file descriptor that's open for reading.

EINTR

The read operation was terminated due to the receipt of a signal, and no data was transferred.

EIO

One of the following:

- A physical I/O error occurred.
- The process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
- (QNX Neutrino extension) The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOMEM

Insufficient space is available.

ENXIO

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EOVERFLOW

The file is a regular file, and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF ) {
            fputc( c, stdout );
        }
        fclose( fp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fgetchar()

Read a character from stdin

Synopsis:

```
#include <stdio.h>

int fgetchar( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fgetchar()* function is the same as [fgetc\(\)](#) (p. 645) with an argument of *stdin*.

Returns:

The next character from *stdin*, cast as `(int)(unsigned char)`, EOF if end-of-file has been reached on *stdin* or if an error occurs ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = freopen( "file", "r", stdin );
    if( fp != NULL ) {
        while( (c = fgetchar()) != EOF ) {
            fputchar(c);
        }
        fclose( fp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fgetpos()

Get the current position of a stream

Synopsis:

```
#include <stdio.h>

int fgetpos( FILE* fp,
             fpos_t* pos );
```

Arguments:

fp

The stream whose position you want to determine.

pos

A pointer to a `fpos_t` object where the function can store the position.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fgetpos()* function stores the current position of the stream *fp* in the `fpos_t` object specified by *pos*.

You can use the value stored in *pos* in a call to [fsetpos\(\)](#) (p. 776) if you want to reposition the file to the position at the time of the *fgetpos()* call.

Returns:

0 for success, or nonzero if an error occurs ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    fpos_t position;
    char buffer[80];

    fp = fopen( "file", "r" );
```

```

    if( fp != NULL ) {
        fgetpos( fp, &position ); /* get position */
        fgets( buffer, 80, fp ); /* read record */
        fsetpos( fp, &position ); /* set position */
        fgets( buffer, 80, fp ); /* read same record */
        fclose( fp );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fgets()

Read a string of characters from a stream

Synopsis:

```
#include <stdio.h>

char* fgets( char* buf,
             size_t n,
             FILE* fp );
```

Arguments:

buf

A pointer to a buffer in which *fgets()* can store the characters that it reads.

n

The maximum number of characters to read.

fp

The stream from which to read the characters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fgets()* function reads a string of characters from the stream specified by *fp*, and stores them in the array specified by *buf*.

It stops reading characters when:

- the end-of-file is reached
- Or:
- a newline (`'\n'`) character is read
- Or:
- *n*-1 characters have been read.

The newline character isn't discarded. A null character is placed immediately after the last character read into the array.



Don't assume that there's a newline character in every string that you read with *fgets()*. A newline character isn't present if there are more than *n*-1 characters before the newline.

Also, a newline character might not appear as the last character in a file when the end-of-file is reached.

Returns:

The same pointer as *buf*, or NULL if the stream is at the end-of-file or an error occurs (*errno* (p. 539) is set).



Use *feof()* (p. 637) or *ferror()* (p. 639) to distinguish an end-of-file condition from an error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    char buffer[80];

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( fgets( buffer, 80, fp ) != NULL ) {
            fputs( buffer, stdout );
        }
        fclose( fp );
    }
    return EXIT_SUCCESS;
}
return EXIT_FAILURE;
```

Classification:

ANSI (p. 46), *POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fgetspent()

Get an entry from the shadow password database

Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

struct spwd* fgetspent( FILE* f );
```

Arguments:

f

The stream from which to read the shadow password database.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fgetspent()* works like the [getspent\(\)](#) (p. 971) function but it assumes that it's reading from a file formatted like a shadow password database file. This function uses a static buffer that's overwritten by each call.



The *fgetspent()*, [getspent\(\)](#) (p. 971), and [getspnam\(\)](#) (p. 974) functions share the same static buffer.

Returns:

A pointer to an object of type `struct spwd` containing the next entry from the password database. For more information about this structure, see [putspent\(\)](#) (p. 2459).

Errors:

The *fgetspent()* function uses the following functions, and as a result *errno* can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)

- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * This program loops, reading a entries from a file
 * (which is formatted in the shadow password way)
 * reading the next shadow password entry.
 * For example this_file /etc/shadow
 */

int main( int argc, char** argv )
{
    FILE*      fp;
    struct spwd* sp;

    if (argc < 2) {
        printf("%s filename \n", argv[0]);
        return(EXIT_FAILURE);
    }

    if (!(fp = fopen(argv[1], "r"))){
        fprintf(stderr, "Can't open file %s \n", argv[1]);
        return(EXIT_FAILURE);
    }

    while( (sp = fgetspent(fp)) != (struct spwd *) 0 ) {
        printf( "Username: %s\n", sp->sp_namp );
        printf( "Password: %s\n", sp->sp_pwdp );
    }

    fclose(fp);
    return( EXIT_SUCCESS );
}
```

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

fgetwc()

Read a wide character from a stream

Synopsis:

```
#include <wchar.h>

wint_t fgetwc( FILE * fp );
```

Arguments:

fp

The stream from which you want to read a character.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fgetwc()* function reads the next wide character from the stream specified by *fp*.

Returns:

The next character from *fp*, cast as `(wint_t)(wchar_t)`, or `WEOF` if end-of-file has been reached or if an error occurs (*errno* (p. 539) is set).



Use *feof()* (p. 637) or *ferror()* (p. 639) to distinguish an end-of-file condition from an error.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *fp* and would have been blocked by this operation.

EBADF

The file descriptor for *fp* isn't valid for reading.

EINTR

A signal terminated the read operation; no data was transferred.

EIO

Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.

EOVERFLOW

Cannot read at or beyond the offset maximum for this stream.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fgetws()

Read a string of wide characters from a stream

Synopsis:

```
#include <wchar.h>

wchar_t * fgetws( wchar_t * buf,
                  int n,
                  FILE * fp );
```

Arguments:

buf

A pointer to a buffer in which *fgetws()* can store the wide characters that it reads.

n

The maximum number of characters to read.

fp

The stream from which to read the characters.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *fgetws()* function reads a string of wide characters from the stream specified by *fp*, and stores them in the array specified by *buf*.

It stops reading wide characters when one of the following occurs:

- The end-of-file is reached.
- A newline (`'\n'`) character is read.
- *n*-1 characters have been read.

The *fgetws()* function places a NUL at the end of the string.



Don't assume all strings have newline characters. A newline character isn't present when more than $n-1$ characters occur before the newline.

Also, a newline character might not appear as the last character in a file when the end-of-file is reached.

Returns:

NULL

Failure; the stream is at the end-of-file or an error occurred ([errno](#) (p. 539) is set).

buf

Success.



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *fp* and would have been blocked by this operation.

EBADF

The file descriptor for *fp* isn't valid for reading.

EINTR

A signal terminated the read operation; no data was transferred.

EIO

Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.

E_OVERFLOW

Cannot read at or beyond the offset maximum for this stream.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fileno()

Return the file descriptor for a stream

Synopsis:

```
#include <stdio.h>

int fileno( FILE * stream );
```

Arguments:

stream

The stream whose file descriptor you want to find.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fileno()* function returns the file descriptor for the specified file *stream*. This file descriptor can be used in POSIX input/output calls anywhere the value returned by [open\(\)](#) (p. 1905) can be used.

To associate a stream with a file descriptor, call [fdopen\(\)](#) (p. 635).



In QNX Neutrino, the file descriptor is also the connection ID (*coid*) used by various QNX Neutrino-specific functions.

The following symbolic values in `<unistd.h>` define the file descriptors associated with the C language [stdin](#) (p. 3176), [stdout](#) (p. 3177), and [stderr](#) (p. 3175) streams:

STDIN_FILENO

Standard input file number, *stdin* (0)

STDOUT_FILENO

Standard output file number, *stdout* (1)

STDERR_FILENO

Standard error file number, *stderr* (2)

Returns:

A file descriptor, or -1 if an error occurs ([errno](#) (p. 539) is set).

Examples:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    FILE *stream;

    stream = fopen( "file", "r" );
    if( stream != NULL ) {
        printf( "File number is %d.\n", fileno( stream ) );
        fclose( stream );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Produces output similar to:

File number is 7.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

flink()

Assign a pathname to a file descriptor

Synopsis:

```
#include <unistd.h>

int flink( int fd,
           const char *path );
```

Arguments:

fd

The file descriptor.

path

The path you want to associate with the file descriptor.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *flink()* function assigns the pathname, *path*, to the file associated with the file descriptor, *fd*.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EACCES

A component of either path prefix denies search permission, or the link named by *path* is in a directory with a mode that denies write permission.

EBADF

The file descriptor *fd* is invalid.

EEXIST

The link named by *path* already exists.

ELOOP

Too many levels of symbolic links or prefixes.

EMLINK

The number of links to the file would exceed `LINK_MAX`.

ENAMETOOLONG

The length of the *path* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENOENT

This error code can mean the following:

- A component of either path prefix doesn't exist.
- The *path* points to an empty string.

ENOSPC

The directory that would contain the link can't be extended.

ENOSYS

The *flink()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of either path prefix isn't a directory.

EROFS

The requested link requires writing in a directory on a read-only file system.

EXDEV

The link named by *path* is on a different logical disk.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

flock()

Apply or remove an advisory lock on an open file

Synopsis:

```
#include <fcntl.h>

int flock( int filedes,
           int operation );
```

Arguments:

filedes

The file descriptor of an open file.

operation

What you want to do to the file; one of the following:

- LOCK_EX—apply an exclusive lock.
- LOCK_SH—apply a shared lock.
- LOCK_UN—unlock an existing lock.

You can OR LOCK_NB with LOCK_EX or LOCK_SH to prevent the function from blocking (see below).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *flock()* function applies or removes an advisory lock on the file associated with the open file descriptor *filedes*.

Advisory locks allow cooperating processes to perform consistent operations on files, but they don't guarantee consistency.

The locking mechanism allows two types of locks: *shared* and *exclusive*. At any time, multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be upgraded to an exclusive lock, and vice versa, by specifying the appropriate lock type. The previous lock is released and a new lock is applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that's already locked causes the caller to be blocked until the lock may be acquired. If you don't want the caller to be blocked, you can specify `LOCK_NB` in the operation to make the call fail instead (*errno* is set to `EWOULDBLOCK`).



Locks are applied to files, not file descriptors. That is, file descriptors duplicated through `dup()` (p. 483) or `fork()` (p. 685) don't result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks, and the child explicitly unlocks the file, the parent loses its lock.

Processes blocked awaiting a lock may be awakened by signals.

Returns:

0

The operation was successful.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EBADF

Invalid descriptor, *filedes*.

EINVAL

The argument operation doesn't include one of `LOCK_EX`, `LOCK_SH`, or `LOCK_UN`.

ENOMEM

The system can't allocate sufficient memory to store lock resources.

ENOSYS

The filesystem doesn't support the operation.

EOPNOTSUPP

The *filedes* argument refers to an object other than a file.

EWOULDBLOCK

The file is already locked, and LOCK_NB was specified.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

flockfile()

Acquire ownership of a file

Synopsis:

```
#include <stdio.h>

void flockfile( FILE* file );
```

Arguments:

file

A pointer to the FILE object for the file you want to lock.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *flockfile()* function provides for explicit application-level locking of *stdio* (FILE) objects. This function can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The *flockfile()* function is used by a thread to acquire ownership of a FILE. The [ftrylockfile\(\)](#) (p. 796) function is a nonblocking version of *flockfile()*. To unlock the file, call [funlockfile\(\)](#) (p. 801).

The implementation acts as if there is a lock count associated with each FILE. This count is implicitly initialized to zero when the FILE is created. The FILE object is unlocked when the count is zero. When the count is positive, a single thread owns the FILE. When the *flockfile()* function is called, if the count is zero or if the count is positive and the caller owns the FILE, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero.

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

floor(), floorf()

Round down a value to the next integer

Synopsis:

```
#include <math.h>

double floor( double x );
float floorf( float x );
```

Arguments:

x

The value you want to round.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the largest integer *x* (rounding towards the “floor”).

Returns:

The largest integer *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", floor( -3.14 ) );
    printf( "%f\n", floor( -3. ) );
    printf( "%f\n", floor( 0. ) );
    printf( "%f\n", floor( 3.14 ) );
    printf( "%f\n", floor( 3. ) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
-4.000000  
-3.000000  
0.000000  
3.000000  
3.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

flushall()*Flush all input/output buffers***Synopsis:**

```
#include <stdio.h>

int flushall( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *flushall()* function flushes all buffers associated with open input/output streams. A subsequent read operation on an input stream reads new data from the associated stream.

Calling the *flushall()* function is equivalent to calling *fflush()* (p. 641) for all open streams.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Classification:

QNX 4 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The QNX 4 version of this function returns the number of streams flushed.

fmod(), fmodf(), fmodl()

Compute a residue, using floating-point modular arithmetic

Synopsis:

```
#include <math.h>

double fmod( double x,
             double y );

float fmodf( float x,
             float y );

long double fmodl( long double x,
                   long double y );
```

Arguments:***x***

An arbitrary number.

y

The modulus.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *fmod()* and *fmodf()* functions compute the floating-point residue of $x \pmod{y}$, which is the remainder of x / y , even if the quotient x / y isn't representable.

Returns:

The residue, $x - (i \times y)$, for some integer i such that, if y is nonzero, the result has the same sign as x and a magnitude less than the magnitude of y .

For a correct value that would cause an underflow, these functions return 0.0. The return value when y is zero is NaN. The return value when x is infinite is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [*errno*](#) (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", fmod( 4.5, 2.0 ) );
    printf( "%f\n", fmod( -4.5, 2.0 ) );
    printf( "%f\n", fmod( 4.5, -2.0 ) );
    printf( "%f\n", fmod( -4.5, -2.0 ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0.500000
-0.500000
0.500000
-0.500000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fnmatch()

Check to see if a file or path name matches a pattern

Synopsis:

```
#include <fnmatch.h>

int fnmatch( const char* pat,
             const char* str,
             int flags );
```

Arguments:

pat

The pattern to match; see “[Pattern Matching Special Characters](#) (p. 678),” below.

str

The string to match against the pattern.

flags

Flags that modify interpretation of *pat* and *str*; a bitwise inclusive OR of these bits:

FNМ_PATHNAME

If this is set, a slash character in *str* is explicitly matched by a slash in *pat*; it isn't matched by either the asterisk or question mark special characters, or by a bracket expression.

FNМ_PERIOD

If this is set, a leading period in *str* matches a period in *pat*, where the definition of “leading” depends on **FNМ_PATHNAME**:

- If **FNМ_PATHNAME** is set, a period is leading if it's the first character in *str*, or if it immediately follows a slash.
- If **FNМ_PATHNAME** isn't set, a period is leading only if it's the first character in *str*.

FNМ_QUOTE

If this isn't set, a backslash (\) in *pat* followed by another character matches that second character. If `FNM_QUOTE` is set, a backslash is treated as an ordinary character.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *fnmatch()* function checks the file or path name specified by the *str* argument to see if it matches the pattern specified by the *pat* argument.

Pattern Matching Special Characters

A pattern-matching special character that is quoted is a pattern that matches the special character itself. When not quoted, such special characters have special meaning in the specification of patterns. The pattern-matching special characters and the contexts in which they have their special meaning are as follows:

?

Matches any printable or nonprintable collating element except <newline>.

Matches any string, including the null string.

[*bracket_expr*]

Matches a single collating element as per Regular Expression Bracket Expressions (1003.2 2.9.1.2) except that:

- The exclamation point character (!) replaces the circumflex character (^) in its role as a nonmatching list in the regular expression notation.
- The backslash is used as an escape character within bracket expressions.

The `?`, `*` and `[` characters aren't special when used inside a bracket expression.

The concatenation of patterns matching a single character is a valid pattern that matches the concatenation of the single characters or collating elements matched by each of the concatenated patterns. For example, the pattern `a[bc]` matches the strings `ab` and `ac`.

The concatenation of one or more patterns matching a single character with one or more asterisks (*) is a valid pattern. In such patterns, each asterisk matches a string of zero or more characters, up to the first character that matches the character following the asterisk in the pattern. For example, the pattern `a*d` matches the strings `ad`, `abd`, and `abcd`, but not the string `abc`.

When an asterisk is the first or last character in a pattern, it matches zero or more characters that precede or follow the characters matched by the remainder of the pattern. For example, the pattern `a*d*` matches the strings `ad`, `abcd`, `abcdef`, `aaaad` and `adddd`; the pattern `*a*d` matches the strings `ad`, `abcd`, `efabcd`, `aaaad` and `adddd`.

Returns:

0

The *str* argument matches the pattern specified by *pat*.

Nonzero

The *str* argument doesn't match the pattern specified by *pat*.

Examples:

```
/*
 * The following example accepts a set of patterns
 * for filenames as argv[1..argc]. It reads lines
 * from standard input, and outputs the lines that
 * match any of the patterns.
 */
#include <stdio.h>
#include <fnmatch.h>
#include <stdlib.h>
#include <limits.h>

int main( int argc, char **argv )
{
    int i;
    char buffer[PATH_MAX+1];

    while( gets( buffer ) ) {
        for( i = 0; i < argc; i++ ) {
            if( fnmatch( argv[i], buffer, 0 ) == 0 ) {
                puts( buffer );
                break;
            }
        }
    }
    exit( EXIT_SUCCESS );
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fopen()*, *fopen64()

Open a file stream

Synopsis:

```
#include <stdio.h>

FILE * fopen( const char * filename,
              const char * mode );

FILE * fopen64( const char * filename,
                const char * mode );
```

Arguments:

filename

The name of the file that you want to open.

mode

The access mode; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fopen()* and *fopen64()* functions open a file stream for the file specified by *filename*. The *fopen64()* function is a 64-bit version of *fopen()*. The *mode* string begins with one of the following sequences:

a

Append: create a new file or open the file for writing at its end.

a+

Append: open the file or create it for update, writing at end-of-file; use the default file translation.

r

Open the file for reading.

r+

Open the file for update (reading and/or writing); use the default file translation.

w

Create the file for writing, or truncate it to zero length.

w+

Create the file for update, or truncate it to zero length; use the default file translation.

You can add the letter `b` to the end of any of the above sequences to indicate that the file is (or must be) a binary file (this is an ANSI requirement for portability to systems that make a distinction between text and binary files, such as DOS). Under QNX Neutrino, there's no difference between text files and binary files.

- Opening a file in read mode (`r` in the *mode*) fails if the file doesn't exist or can't be read.
- Opening a file in append mode (`a` in the *mode*) causes all subsequent writes to the file to be forced to the current end-of-file, regardless of previous calls to the [fseek\(\)](#) (p. 772) function.
- When a file is opened with update mode (`+` in the *mode*), both input and output may be performed on the associated stream.



When using a stream in update mode, writing can't be followed by reading without an intervening call to [fflush\(\)](#) (p. 641), or to a file-positioning function ([fseek\(\)](#) (p. 772), [fsetpos\(\)](#) (p. 776) or [rewind\(\)](#) (p. 2646)). Similarly, reading can't be followed by writing without an intervening call to a file-positioning function, unless the read resulted in end-of-file.

The largest value that can be represented correctly in an object of type `off_t` shall be established as the offset maximum in the open file description.

Returns:

A pointer to a file stream for success, or `NULL` if an error occurs ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied on a component of the *filename* prefix, or the file exists and the permissions specified by *mode* are denied, or the file doesn't exist and write permission is denied for the parent directory of the file to be created.

EBADFSYS

While attempting to open the named file, either the file itself or a component of the *filename* prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You'll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY

File access was denied due to a conflicting open (see [sopen\(\)](#) (p. 3083)).

EINTR

The *fopen()* operation was interrupted by a signal.

EINVAL

The value of the *mode* argument is not valid.

EISDIR

The named file is a directory, and the *mode* argument specifies write-only or read/write access.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the *filename* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENFILE

Too many files are currently open in the system.

ENOENT

Either the named file or the *filename* prefix doesn't exist, or the *filename* argument points to an empty string.

ENOMEM

There isn't enough memory for the `FILE` structure.

ENOSPC

The directory or filesystem that would contain the new file can't be extended.

ENOSYS

The *fopen()* function isn't implemented for the filesystem specified in *filename*.

ENOTDIR

A component of the *filename* prefix isn't a directory.

ENXIO

The media associated with the file has been removed (e.g. CD, floppy).

EOVERFLOW

The named file is a regular file and the size of the file can't be represented correctly in an object of type *off_t*.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;

    fp = fopen( "report.dat", "r" );
    if( fp != NULL ) {
        /* rest of code goes here */
        fclose( fp );

        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

fopen() is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *fopen64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fork()

Create a new process

Synopsis:

```
#include <sys/types.h>
#include <process.h>

pid_t fork( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fork()* function creates a new process. The new process (child process) is an exact copy of the calling process (parent process), except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (which is the process ID of the calling process).
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors refers to the same open file description with the corresponding file descriptor of the parent.
- The child process has its own copy of the parent's open directory streams.
- The child process's values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are set to zero.
- The child process has a single thread, regardless of the number that the parent process has.
- File locks previously set by the parent aren't inherited by the child.
- Pending alarms are cleared for the child process.
- The set of signals pending for the child process is initialized to the empty set.
- The child process doesn't inherit any memory locks that the parent set. For more information, see “Locking memory” in the Process Manager chapter of the *System Architecture* guide.

In order to successfully call this function, your process must have the `PROCMGR_AID_FORK` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The parent's ability configuration is copied to the child verbatim, regardless of the `PROCmgr_AOP_INHERIT_NO` status of each of the abilities.

You can use [pthread_atfork\(\)](#) (p. 2187) to register fork handler functions to be called before or after the fork occurs.



If your program uses mutexes, you need to register a `pthread_atfork()` handler that locks all the mutexes before you fork.

Returns:

A value of zero to the child process; and the process ID of the child process to the parent process. Both processes continue to execute from the `fork()` function. If an error occurs, `fork()` returns -1 to the parent and sets [errno](#) (p. 539).

Errors:

EAGAIN

Insufficient resources are available to create the child process. For example, you might have exceeded the maximum number of processes permitted; see the `RLIMIT_NPROC` resource for [getrlimit\(\)](#) (p. 942).

ENOMEM

The process requires more memory than the system is able to supply.

ENOSYS

The `fork()` function isn't implemented for this memory protection model.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

```
/*
 * This program executes the program and arguments
 * specified by argv[1..argc]. The standard input
 * of the executed program is converted to upper
 * case.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <process.h>
#include <sys/wait.h>

int main( int argc, char **argv )
{
    pid_t pid;
    pid_t wpid;
    int    fd[2];
```

```

char  buffer[80];
int   i, len;
int   status;

if( pipe( fd ) == -1 ) {
    perror( "pipe" );
    return EXIT_FAILURE;
}

if( ( pid = fork() ) == -1 ) {
    perror( "fork" );
    return EXIT_FAILURE;
}

if( pid == 0 ) {
    /* This is the child process.
     * Move read end of the pipe to stdin ( 0 ),
     * close any extraneous file descriptors,
     * then use exec to 'become' the command.
     */
    dup2( fd[0], 0 );
    close( fd[1] );
    execvp( argv[1], argv+1 );

/* This can only happen if exec fails; print message
 * and exit.
 */
    perror( argv[1] );
    return EXIT_FAILURE;
} else {
    /* This is the parent process.
     * Remove extraneous file descriptors,
     * read descriptor 0, write into pipe,
     * close pipe, and wait for child to die.
     */
    close( fd[0] );
    while( ( len = read( 0, buffer, sizeof( buffer ) )
            ) > 0 ) {
        for( i = 0; i < len; i++ ) {
            if( isupper( buffer[i] ) )
                buffer[i] = tolower( buffer[i] );
        }
        write( fd[1], buffer, len );
    }
    close( fd[1] );
    do {
        wpid = waitpid( pid, &status, 0 );
    } while( WIFEXITED( status ) == 0 );
    return WEXITSTATUS( status );
}
}

```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

forkpty()

Create a new process operating in a pseudo-tty

Synopsis:

```
#include <unistd.h>

pid_t forkpty( int *amaster,
               char *name,
               struct termios *termp,
               struct winsize *winp );
```

Arguments:

amaster

A pointer to a location where *forkpty()* can store the file descriptor of the master side of the pseudo-tty.

name

NULL, or a pointer to a buffer where *forkpty()* can store the filename of the slave side of the pseudo-tty.

termp

NULL, or a pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes to apply to the slave side of the pseudo-tty.

winp

A pointer to a `winsize` structure that defines the window size to use for the slave side of the pseudo-tty.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *forkpty()* function combines *openpty()* (p. 1921), *fork()* (p. 685), and *login_tty()* (p. 1547) to create a new process operating in a pseudo-tty.

This function fails if either *openpty()* or *fork()* fails.

Returns:

0 to the child process, the child's process ID to the parent, or -1 if an error occurred.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fp_exception_mask()

Get or set the current exception mask

Synopsis:

```
#include <fpstatus.h>

int fp_exception_mask ( int new_mask,
                       int set );
```

Arguments:

new_mask

The new mask to apply. The bits include:

- `_FP_EXC_INVALID`
- `_FP_EXC_DIVZERO`
- `_FP_EXC_OVERFLOW`
- `_FP_EXC_UNDERFLOW`
- `_FP_EXC_INEXACT`
- `_FP_EXC_DENORMAL`

set

A value that indicates what you want the function to do:

- If *set* < 0, return the current mask. The *new_mask* argument is ignored.
- If *set* = 0, disable the bits in the exception mask that correspond to the bits set in *new_mask*.
- If *set* > 0, enable the bits in the exception mask that correspond to the bits set in *new_mask*.

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The *fp_exception_mask()* function gets or sets the current exception mask, depending on the value of the *set* argument.

Returns:**If *set* < 0**

The current exception mask.

If *set* = 0

The previous mask.



This function doesn't return a special value to indicate that an error occurred. If you want to check for errors, set *errno* (p. 539) to 0, call the function, and then check *errno* again.

Examples:

```
#include <fpstatus.h>

int main(int argc, char** argv)
{
    int ret;

    if ((ret = fp_exception_mask(0, -1)) < 0)
        printf("*** Problem retrieving exceptions \n");
    printf("Exceptions Enabled: \n\t");
    if (ret & _FP_EXC_INEXACT)
        printf("Inexact ");
    if (ret & _FP_EXC_DIVZERO)
        printf("DivZero ");
    if (ret & _FP_EXC_UNDERFLOW)
        printf("Underflow ");
    if (ret & _FP_EXC_OVERFLOW)
        printf("Overflow ");
    if (ret & _FP_EXC_INVALID)
        printf("Invalid ");
    printf("\n");

    /* Set the exception mask to enable division by zero errors */
    if ((ret = fp_exception_mask(_FP_EXC_DIVZERO, 1)) < 0)
        printf("*** Problem setting exceptions \n");
    if ((ret = fp_exception_mask(0, -1)) < 0)
        printf("*** Problem retrieving exceptions \n");
    printf("Exceptions Enabled: \n\t");
    if (ret & _FP_EXC_INEXACT)
        printf("Inexact ");
    if (ret & _FP_EXC_DIVZERO)
        printf("DivZero ");
    if (ret & _FP_EXC_UNDERFLOW)
        printf("Underflow ");
    if (ret & _FP_EXC_OVERFLOW)
        printf("Overflow ");
    if (ret & _FP_EXC_INVALID)
        printf("Invalid ");
    printf("\n");

    return(0);
}
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fp_exception_value()

Get the value of the current exception registers

Synopsis:

```
#include <fpstatus.h>

int fp_exception_value( int mask );
```

Arguments:

mask

A mask whose bits indicate which registers you want the value of. The bits include:

- `_FP_EXC_INVALID`
- `_FP_EXC_DIVZERO`
- `_FP_EXC_OVERFLOW`
- `_FP_EXC_UNDERFLOW`
- `_FP_EXC_INEXACT`
- `_FP_EXC_DENORMAL`

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `fp_exception_value()` function gets the value of the current exception registers. Set bits indicate that the exception has signaled, unset bits indicate that the exception hasn't signaled.

Returns:

The value of the current exception registers based on the values from `<fpstatus.h>`.



This function doesn't return a special value to indicate that an error occurred. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check `errno` again.

Examples:

```
#include <fpstatus.h>

int main(int argc, char** argv)
{
    int ret;

    /* Test to see if an operation has set (but not necessarily
     * signaled depending on the exception mask) the
     * division by zero bit:
     */

    if (fp_exception_value(_FP_EXC_DIVZERO) & _FP_EXC_DIVZERO)
        printf("Division by zero has occurred \n");
    else
        printf("Division by zero has not occurred \n");

    return(0);
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fp_precision()

Set or get the current precision

Synopsis:

```
#include <fpstatus.h>

int fp_precision( int newprecision );
```

Arguments:

newprecision

The new precision; one of:

- < 0 — return the current setting.
- _FP_PREC_FLOAT
- _FP_PREC_DOUBLE
- _FP_PREC_EXTENDED
- _FP_PREC_DOUBLE_EXTENDED

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *fp_precision()* function sets or gets the current floating-point precision, depending on the value of *newprecision*.

Returns:

If *newprecision* is less than 0, the current precision; otherwise, the previous precision.



This function doesn't return a special value to indicate that an error occurred. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again.

Examples:

```
#include <fpstatus.h>

int main(int argc, char** argv)
{
```

```
int ret;

ret = fp_precision(-1);
printf("Precision: ");
if (ret == _FP_PREC_FLOAT)
    printf("Float \n");
else if (ret == _FP_PREC_DOUBLE)
    printf("Double \n");
else if (ret == _FP_PREC_EXTENDED)
    printf("Extended \n");
else if (ret == _FP_PREC_DOUBLE_EXTENDED)
    printf("128 Bit \n");
else if (ret == _FP_PREC_EXTENDED)
    printf("Extended \n");
else if (ret == _FP_PREC_DOUBLE_EXTENDED)
    printf("128 Bit \n");
else
    printf("Error \n");

return(0);
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fp_rounding()

Set or get the current rounding

Synopsis:

```
#include <fpstatus.h>

int fp_rounding( int newrounding );
```

Arguments:

newrounding

The new rounding; one of:

- < 0 — return the current setting.
- _FP_ROUND_NEAREST
- _FP_ROUND_ZERO
- _FP_ROUND_POSITIVE
- _FP_ROUND_NEGATIVE

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *fp_rounding()* function sets or gets the current rounding mode, depending on the value of *newrounding*.

Returns:

If *newrounding* is less than 0, the current rounding mode; otherwise, the previous mode.



This function doesn't return a special value to indicate that an error occurred. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again.

Examples:

```
#include <fpstatus.h>
#include <stdlib.h>
```

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int ret;

    ret = fp_rounding(-1);
    printf("Rounding mode: ");
    if (ret == _FP_ROUND_NEAREST)
        printf("Nearest \n");
    else if (ret == _FP_ROUND_POSITIVE)
        printf("Positive \n");
    else if (ret == _FP_ROUND_NEGATIVE)
        printf("Negative \n");
    else if (ret == _FP_ROUND_ZERO)
        printf("To Zero \n");
    else
        printf("Error \n");

    return EXIT_SUCCESS;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fpathconf()

Return the value of a configurable limit associated with a file

Synopsis:

```
#include <unistd.h>

long fpathconf( int filedes,
                int name );
```

Arguments:

filedes

A file descriptor for the file whose limit you want to check.

name

The name of the configurable limit; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fpathconf()* function returns a value of a configurable limit indicated by *name* that's associated with the file indicated by *filedes*.

Configurable limits are defined in `<confname.h>`, and include at least the following values:

`_PC_ASYNC_IO`

Defined (not -1) if asynchronous I/O is supported for the file.

`_PC_CHOWN_RESTRICTED`

If defined (not -1), indicates that the use of the *chown()* function is restricted to a process with appropriate privileges, and to changing the group ID of a file to the effective group ID of the process or to one of its supplementary group IDs.

`_PC_LINK_DIR`

Defined (not -1) if the filesystem permits the unlinking of a directory.

`_PC_LINK_MAX`

Maximum value of a file's link count.

`_PC_MAX_CANON`

Maximum number of bytes in a terminal's canonical input buffer (edit buffer).

`_PC_MAX_INPUT`

Maximum number of bytes in a terminal's raw input buffer.

`_PC_NAME_MAX`

Maximum number of bytes in a file name (not including the terminating null).

`_PC_NO_TRUNC`

If defined (not -1), indicates that the use of pathname components longer than the value given by `_PC_NAME_MAX` generates an error.

`_PC_PATH_MAX`

Maximum number of bytes in a pathname (not including the terminating null).

`_PC_PIPE_BUF`

Maximum number of bytes that can be written atomically when writing to a pipe.

`_PC_PRIO_IO`

Defined (not -1) if prioritized I/O is supported for the file.

`_PC_SYNC_IO`

Defined (not -1) if synchronous I/O is supported for the file.

`_PC_VDISABLE`

If defined (not -1), this is the character value that can be used to individually disable special control characters in the [termios](#) (p. 3379) control structure.

Returns:

The requested configurable limit, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

`EINVAL`

The *name* argument is invalid, or the indicated limit isn't supported for this *files*.

EBADF

The argument *files* is invalid.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    long value;

    value = fpathconf( 0, _PC_MAX_INPUT );
    printf( "Input buffer size is %ld bytes\n",
        value );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fprintf()

Write output to a stream

Synopsis:

```
#include <stdio.h>

int fprintf( FILE* fp,
             const char* format,
             ... );
```

Arguments:

fp

The stream to which you want to send the output.

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fprintf()* function writes output to the stream specified by *fp*, under control of the *format* specifier.

Returns:

The number of characters written, or a negative value if an output error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

char *weekday = { "Saturday" };
char *month = { "April" };

int main( void )
{
```

```
    fprintf( stdout, "%s, %s %d, %d\n",  
            weekday, month, 10, 1999 );  
    return EXIT_SUCCESS;  
}
```

Produces:

Saturday, April 10, 1999

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fputc()

Write a character to a stream

Synopsis:

```
#include <stdio.h>

int fputc( int c,
           FILE* fp );
```

Arguments:

c

The character you want to write.

fp

The stream you want to write the character to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fputc()* function writes the character specified by *c*, cast as `(int)(unsigned char)`, to the stream specified by *fp*.

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor underlying *fp*, and the process would be delayed in the write operation.

EBADF

The file descriptor underlying *fp* isn't a valid file descriptor that's open for writing.

EFBIG

One of the following:

- An attempt was made to write a file that exceeds the maximum file size.
- An attempt was made to write a file that exceeds the process's file size limit.
- The file is a regular file, and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EINTR

The write operation was terminated due to the receipt of a signal, and no data was transferred.

EIO

One of the following:

- A physical I/O error occurred.
- The process is in a background process group attempting to write to its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
- (QNX Neutrino extension) The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOMEM

Insufficient space is available.

ENXIO

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

ENOSPC

There was no free space remaining on the device containing the file.

EPIPE

An attempt was made to write to a pipe or FIFO that wasn't open for reading by any process. A SIGPIPE signal is also sent to the thread.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF ) {
            fputc( c, stdout );
        }
        fclose( fp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *c* is negative, the value returned by this function isn't equal to *c* — unless *c* is -1 and an error occurred :-)

fputc()

Write a character to stdout

Synopsis:

```
#include <stdio.h>

int fputc( int c );
```

Arguments:

c

The character you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fputc()* function writes the character specified by *c*, cast as `(int)(unsigned char)`, to *stdout*. It's equivalent to [putc\(\)](#) (p. 2451) and to:

```
fputc( c, stdout );
```

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        c = fgetc( fp );
        while( c != EOF ) {
            fputc( c );
            c = fgetc( fp );
        }
        fclose( fp );
    }
```

```
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *c* is negative, the value returned by this function isn't equal to *c* — unless *c* is -1 and an error occurred :-)

fputs()

Write a string to an output stream

Synopsis:

```
#include <stdio.h>

int fputs( const char* buf,
           FILE* fp );
```

Arguments:

buf

The string you want to write.

fp

The stream you want to write the string to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fputs()` function writes the character string specified by *buf* to the output stream specified by *fp*.



The terminating NUL character isn't written.

Returns:

A nonnegative value for success, or EOF if an error occurs ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp_in, *fp_out;
    char buffer[80];
```

```
    fp_in = fopen( "file", "r" );
    fp_out = fopen( "outfile", "w" );
    if( fp_in != NULL && fp_out != NULL ) {
        while( fgets( buffer, 80, fp_in ) != NULL ) {
            fputs( buffer, fp_out );
        }
        fclose( fp_in );
        fclose( fp_out );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fputwc()

Write a wide character to a stream

Synopsis:

```
#include <wchar.h>

wint_t fputwc( wchar_t wc,
               FILE * fp );
```

Arguments:

wc

The wide character you want to write.

fp

The stream you want to write the character to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fputwc()` function writes the wide character specified by `wc`, cast as `(wint_t)(wchar_t)`, to the stream specified by `fp`.

Returns:

The wide character written, cast as `(wint_t)(wchar_t)`, or `WEOF` if an error occurred ([errno](#) (p. 539) is set).



If `wc` exceeds the valid wide-character range, the value returned is the wide character written, not `wc`.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for `fp` and would have been blocked by this operation.

EBADF

The stream specified by *fp* isn't valid for writing.

EFBIG

The file exceeds the maximum file size, the process's file size limit, or the function can't write at or beyond the offset maximum.

EINTR

A signal terminated the write operation; no data was transferred.

EIO

A physical I/O error has occurred or all of the following conditions were met:

- The process is in the background.
- **TOSTOP** is set.
- The process is blocking/ignoring **SIGTTOU**.
- The process group is orphaned.

EPIPE

Can't write to pipe or FIFO because it's closed; a **SIGPIPE** signal is also sent to the thread.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fputws()

Write a wide-character string to an output stream

Synopsis:

```
#include <wchar.h>

int fputws( const wchar_t * ws,
            FILE * fp );
```

Arguments:

buf

The wide-character string you want to write.

fp

The stream you want to write the string to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fputws()* function writes the wide-character string specified by *ws* to the output stream specified by *fp*.



The terminating NUL wide character isn't written.

Returns:

A nonnegative value for success, or `WEOF` if an error occurs ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *fp* and would have been blocked by this operation.

EBADF

The stream specified by *fp* isn't valid for writing.

EFBIG

The file exceeds the maximum file size, the process's file size limit, or the function can't write at or beyond the offset maximum.

EINTR

A signal terminated the write operation; no data was transferred.

EIO

A physical I/O error has occurred or all of the following conditions were met:

- The process is in the background.
- **TOSTOP** is set.
- The process is blocking/ignoring **SIGTTOU**.
- The process group is orphaned.

EPIPE

Can't write to pipe or FIFO because it's closed; a **SIGPIPE** signal is also sent to the thread.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fread()

Read elements of a given size from a stream

Synopsis:

```
#include <stdio.h>

size_t fread( void* buf,
              size_t size,
              size_t num,
              FILE* fp );
```

Arguments:

buf

A pointer to a buffer where the function can store the elements that it reads.

size

The size of each element to read.

num

The number of elements to read.

fp

The stream from which to read the elements.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fread()* function reads *num* elements of *size* bytes each from the stream specified by *fp* into the buffer specified by *buf*.

Returns:

The number of complete elements successfully read; this value may be less than the requested number of elements.

Use the [feof\(\)](#) (p. 637) and [ferror\(\)](#) (p. 639) functions to determine whether the end of the file was encountered or if an input/output error has occurred.

Errors:

If an error occurs, [errno](#) (p. 539) is set to indicate the type of error.

Examples:

The following example reads a simple student record containing binary data. The student record is described by the struct `student_data` declaration.

```
#include <stdio.h>
#include <stdlib.h>

struct student_data {
    int student_id;
    unsigned char marks[10];
};

size_t read_data( FILE *fp, struct student_data *p )
{
    return( fread( p, sizeof( struct student_data ), 1, fp ) );
}

int main( void )
{
    FILE *fp;
    struct student_data std;
    int i;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( read_data( fp, &std ) != 0 ) {
            printf( "id=%d ", std.student_id );

            for( i = 0; i < 10; i++ ) {
                printf( "%3d ", std.marks[ i ] );
            }

            printf( "\n" );

            fclose( fp );

            return EXIT_SUCCESS;
        }

        return EXIT_FAILURE;
    }
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

free()

Deallocate a block of memory

Synopsis:

```
#include <stdlib.h>

void free( void* ptr );
```

Arguments:

ptr

A pointer to the block of memory that you want to free. It's safe to call *free()* with a NULL pointer.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *free()* function deallocates the memory block specified by *ptr*, which was previously returned by a call to [calloc\(\)](#) (p. 255), [malloc\(\)](#) (p. 1569) or [realloc\(\)](#) (p. 2527).

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main( void )
{
    char *buffer;

    buffer = (char *)malloc( 80 );
    if( buffer == NULL ) {
        printf( "Unable to allocate memory\n" );
        return EXIT_FAILURE;
    } else {
        /* rest of code goes here */

        free( buffer ); /* deallocate buffer */
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Calling *free()* on a pointer already deallocated by a call to *free()* or *realloc()* could corrupt the memory allocator's data structures.

freeaddrinfo()

Free a list of address information structures

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo( struct addrinfo * ai );
```

Arguments:

ai

A pointer to the [addrinfo](#) (p. 122) structure that's at the beginning of the list to be freed.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *freeaddrinfo()* function frees the given list of `addrinfo` structures and the dynamic storage associated with each item in the list.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

freeifaddrs()

Free a network interface address

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>

void freeifaddrs( struct ifaddrs * ifap );
```

Arguments:

ifap

A pointer to the linked list of *ifaddrs* structures to be freed.

Library:

libsocket
Use the -l socket option to qcc to link against this library.

Description:

The *freeifaddrs()* function frees the dynamically allocated data returned by [getifaddrs\(\)](#) (p. 885).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

freopen()*, *freopen64()

Reopen a stream

Synopsis:

```
#include <stdio.h>

FILE* freopen( const char* filename,
              const char* mode,
              FILE* fp );

FILE* freopen64( const char* filename,
                const char* mode,
                FILE* fp );
```

Arguments:

filename

The name of the file to open.

mode

The mode to use when opening the file. For more information, see [fopen\(\)](#) (p. 681).

fp

The stream to associate with the file.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *freopen()* and *freopen64()* functions close the open stream *fp*, open the file specified by *filename*, and associate its stream with *fp*. The *freopen64()* function is a 64-bit version of *freopen()*.

The largest value that can be represented correctly in an object of type `off_t` is the offset maximum in the open file description.

(QNX Neutrino extension) The following mode changes are permitted:

- `w` to `a`

- a to w
- r+ to r
- r+ to w
- r+ to a
- r+ to w+
- r+ to a+
- w+ to r
- w+ to w
- w+ to a
- w+ to r+
- w+ to w+
- w+ to a+
- a+ to r
- a+ to w
- a+ to a
- a+ to r+
- a+ to w+

Returns:

A pointer to the newly opened stream, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied on a component of the *filename* prefix, or the file exists and the permissions specified by *mode* are denied, or the file doesn't exist and write permission is denied for the parent directory of the file to be created.

EBADF

The underlying file descriptor is invalid or doesn't support the requested mode change.

EBADFSYS

While attempting to open the named file, either the file itself or a component of the *filename* prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You'll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY

File access was denied due to a conflicting open (see [sopen\(\)](#) (p. 3083)).

EINTR

The *freopen()* operation was interrupted by a signal.

EINVAL

The value of the *mode* argument is not valid.

EISDIR

The named file is a directory, and the *mode* argument specifies write-only or read/write access.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the *filename* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENFILE

Too many files are currently open in the system.

ENOENT

Either the named file or the *filename* prefix doesn't exist, or the *filename* argument points to an empty string.

ENOMEM

There is no memory for FILE structure.

ENOSPC

The directory or filesystem that would contain the new file can't be extended.

ENOSYS

The *freopen()* function isn't implemented for the filesystem specified in *filename*.

ENOTDIR

A component of the *filename* prefix isn't a directory.

ENXIO

The media associated with the file has been removed (e.g. CD, floppy).

EOVERFLOW

The named file is a regular file and the size of the file can't be represented correctly in an object of type `off_t`.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE* fp;
    int c;

    /* Reopen the stdin stream so it's reading
     * from "file" instead of standard input.
     */
    fp = freopen( "file", "r", stdin );

    if( fp != NULL ) {
        /* Now we can read from "file" using the
         * stdin functions like fgetchar()...
         */
        while( ( c = fgetchar() ) != EOF ) {
            fputchar( c );
        }

        fclose( fp );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

freopen() is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *freopen64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

frexp(), frexpf()

Break a floating-point number into a normalized fraction and an integral power of 2

Synopsis:

```
#include <math.h>

double frexp( double value,
              int* exp );

float frexpf( float value,
              int* exp );
```

Arguments:

value

The value you want to break into a normalized fraction.

exp

A pointer to a location where the function can store the integral power of 2.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions break a floating-point number into a normalized fraction and an integral power of 2. It stores the integral power of 2 in the `int` pointed to by *exp*.

Returns:

x, such that *x* is a `double` with magnitude in the interval [0.5, 1] or 0, and *value* equals *x* times 2 raised to the power *exp*. If *value* is 0, then both parts of the result are 0.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    int expon;
    double value;
```

```
        value = frexp( 4.25, &expon );
        printf( "%f %d\n", value, expon );
        value = frexp( -4.25, &expon );
        printf( "%f %d\n", value, expon );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0.531250 3
-0.531250 3
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_check()

Determine if the underlying filesystem supports encryption

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_check( const char *path,
                    int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_check()* function determines if the underlying filesystem supports encryption.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTOREPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTOREPLY_SUPPORTED

The filesystem supports encryption.

FS_CRYPTOREPLY_NOSUPPORT

The filesystem doesn't support encryption.

Returns:

EOK

Success.

EINVAL

The *path* is invalid.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_add()

Create a domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_add( const char *path,
                          int domain,
                          int type,
                          int state,
                          int length,
                          const uint8_t bytes[],
                          int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The domain number to add (the domain must be unused).

type

The type of encryption to employ; one of the following (defined in <fs_crypto.h>):

- FS_CRYPT0_TYPE_NONE
- FS_CRYPT0_TYPE_XTS — AES-256, in XTS mode
- FS_CRYPT0_TYPE_CBC — AES-256, in CBC mode

state

The lock state for the new domain (1 is locked, 0 is unlocked).

length

The number of bytes available in the key array.

bytes

The binary representation of the key.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the `-l fscrypto` option to `qcc` to link against this library.

Description:

The `fs_crypto_domain_add()` function creates the given domain/type if it doesn't already exist. The domain is created in either the unlocked or locked state, as desired.



You must be in the group that owns the filesystem's mountpoint in order to create a domain.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_COMPLETE

The domain was successfully created.

FS_CRYPTO_REPLY_ERRNO

See the error code.

FS_CRYPTO_REPLY_DOMAIN_EXISTS

The domain already exists.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_READONLY

The filesystem is mounted read-only.

FS_CRYPTO_REPLY_UNKNOWN_TYPE

The *type* argument isn't a valid type of encryption.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403), [fs_crypto_domain_unlock\(\)](#) (p. 745), [malloc\(\)](#) (p. 1569), or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_key_change()

Change the master domain key used to encrypt the domain key

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_key_change( const char *path,
                                int domain,
                                int length,
                                const uint8_t org_key[],
                                const uint8_t new_key[],
                                int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The number of the domain that you want to change the key for.

length

The number of bytes available in the key arrays. The keys must both be the same size.

org_key

The binary representation of the original key.

new_key

The binary representation of the new key.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The `fs_crypto_domain_key_change()` function changes the master domain key used to encrypt the domain key.



You must be in the group that owns the filesystem's mountpoint in order to change the key for a domain.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_BAD_KEY

One of the keys was invalid.

FS_CRYPTO_REPLY_COMPLETE

The key was successfully changed.

FS_CRYPTO_REPLY_ERRNO

See the error code for more information.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_UNKNOWN_DOMAIN

There's no domain with the given number.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

ENOMEM

Insufficient free memory.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_key_check()

Determine if a given domain key is valid

Synopsis:

```

#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_key_check( const char *path,
                               int domain,
                               int length,
                               const uint8_t bytes[],
                               int *preply );

```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The number of the domain whose key you want to check.

length

The number of bytes available in the key array.

bytes

The binary representation of the key.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_domain_key_check()* function determines if a given domain key is valid.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPT0_REPLY_BAD_KEY

The key is invalid.

FS_CRYPTO_REPLY_COMPLETE

The key is valid.

FS_CRYPTO_REPLY_ERRNO

See the error code for more information.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_UNKNOWN_DOMAIN

There's no domain with the given number.

FS_CRYPTO_REPLY_UNKNOWN_TYPE

The type of encryption used for the domain is invalid.

Returns:

EOK

Success.

EBADFSYS

An inconsistency was found in the domain key in the filesystem.

EINVAL

Invalid arguments.

ENOMEM

Insufficient free memory.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

fs_crypto_domain_key_size()

Return the size of keys needed by the filesystem encryption

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_key_size( void );
```

Library:

libfscrypto
Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_domain_key_size()* function returns the size of keys needed by the filesystem encryption.

Returns:

The size of the keys, in bytes.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_lock()

Lock a domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_lock( const char *path,
                           int domain,
                           int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The number of the domain that you want to lock.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_domain_lock()* function locks a domain, preventing access to the original contents of any file belonging to the given domain.



You must be in the group that owns the filesystem's mountpoint in order to lock a domain.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_ALREADY

The domain was already locked.

FS_CRYPTO_REPLY_COMPLETE

The domain is now locked.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_NOENTRY

No entry was found for the given domain.

FS_CRYPTO_REPLY_UNKNOWN_TYPE

The type of encryption used for the domain is invalid.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

ENOMEM

Insufficient free memory.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_query()

Get status information for a domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_query( const char *path,
                           int domain,
                           int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The number of the domain that you want the status of.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_domain_query()* function gets status information for a domain.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_LOCKED

The domain is currently locked.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_NOENTRY

No entry was found for the given domain.

FS_CRYPTO_REPLY_UNLOCKED

The domain is currently unlocked.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_remove()

Destroy a domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_remove( const char *path,
                             int domain,
                             int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The number of the domain that you want to remove.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_domain_remove()* function destroys a domain. Once a domain has been removed, the private key data used to encrypt files associated with the domain is lost, and it isn't possible to retrieve any files in the domain.



You must be in the group that owns the filesystem's mountpoint in order to remove a domain. The domain must also be unlocked.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTOREPLY_COMPLETE

The domain was successfully deleted.

FS_CRYPTO_REPLY_ERRNO

See the error code for more information.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_READONLY

The filesystem is mounted read-only.

FS_CRYPTO_REPLY_UNKNOWN_DOMAIN

There's no domain with the given number.

Returns:**EOK**

Success.

EACCES

The calling process doesn't have permission to remove the domain.

EINVAL

Invalid arguments.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_domain_unlock()

Unlock a domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_domain_unlock( const char *path,
                             int domain,
                             int length,
                             const uint8_t bytes[],
                             int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

domain

The number of the domain that you want to unlock.

length

The number of bytes available in the key array.

bytes

The binary representation of the key.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_domain_unlock()* function unlocks a domain.



You must be in the group that owns the filesystem's mountpoint in order to unlock a domain.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_ALREADY

The domain is already unlocked.

FS_CRYPTO_REPLY_BAD_KEY

The key is invalid.

FS_CRYPTO_REPLY_COMPLETE

The domain is now unlocked.

FS_CRYPTO_REPLY_ERRNO

See the error code for more information.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_UNKNOWN_DOMAIN

There's no domain with the given number.

FS_CRYPTO_REPLY_UNKNOWN_TYPE

The type of encryption used for the domain is invalid.

Returns:

EOK

Success.

EBADFSYS

An inconsistency was found in the domain key in the filesystem.

EINVAL

Invalid arguments.

ENOMEM

Insufficient free memory.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_enable()

Enable encryption support on a filesystem

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_enable( const char *path,
                     int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the `-l fscrypto` option to `qcc` to link against this library.

Description:

The *fs_crypto_enable()* function enables encryption support on a volume that wasn't set up for it at formatting time.



You must be in the group that owns the filesystem's mountpoint in order to enable encryption.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_ALREADY

Encryption was already enabled on the filesystem.

FS_CRYPTO_REPLY_COMPLETE

Encryption is now enabled on the filesystem.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTOREPLY_READONLY

The filesystem is read-only.

Returns:

EOK

Success.

EBADFSYS

Corrupted filesystem detected.

ENOTDIR

The path isn't a directory.

ENOTSUP

The filesystem doesn't support encryption.

EROFS

The filesystem is mounted read-only.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403), [open64\(\)](#) (p. 1905), [fstat64\(\)](#) (p. 779), or [fstatvfs64\(\)](#) (p. 783).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_enable_option()

Enable encryption support on a filesystem

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_enable_option( const char *path,
                             uint32_t option,
                             uint32_t domain,
                             int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

option

The option to enable. The only option currently defined is:

- **FS_CRYPT0_ENABLE_WHOLE_DISK** — encrypt the entire disk as one domain, specified by the *domain* argument. All plain-text files are assumed to have been tagged for migration. All plain-text files are hidden so that they can't be viewed or accessed.

domain

The domain (if required for the option).

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_enable_option()* function enables encryption support on a volume that wasn't set up for it at formatting time.



You must be in the group that owns the filesystem's mountpoint in order to enable encryption.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTOREPLY_ALREADY

Encryption was already enabled on the filesystem.

FS_CRYPTOREPLY_COMPLETE

Encryption is now enabled on the filesystem.

FS_CRYPTOREPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTOREPLY_READONLY

The filesystem is read-only.

FS_CRYPTOREPLY_UNKNOWN_DOMAIN

The domain (if required by the option) doesn't exist.

Returns:

EOK

Success.

EBADFSYS

Corrupted filesystem detected.

ENOTDIR

The path isn't a directory.

ENOTSUP

The filesystem doesn't support encryption.

EROFS

The filesystem is mounted read-only.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403), [open64\(\)](#) (p. 1905), [fstat64\(\)](#) (p. 779), or [fstatvfs64\(\)](#) (p. 783).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_file_get_domain()

Get the domain of a file or directory

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_file_get_domain( const char *path,
                              int *pdomain,
                              int *preply );
```

Arguments:

path

The path to the file or directory whose domain you want to get.

domain

A pointer to a location where the function can store the domain.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_file_get_domain()* function gets the domain number for the given file or directory, storing the number in the location that *domain* points to.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_COMPLETE

The function retrieved the domain number.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_NOT_ASSIGNED

The file or directory hasn't been assigned to a domain.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_key_gen()

Generate an encryption key from a password

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_key_gen( const char *password,
                      int plen,
                      const uint8_t *salt,
                      int slen,
                      uint8_t *key,
                      int klen );
```

Arguments:***password***

The password to encrypt.

plen

The number of bytes in the given password string.

salt

The bytes to use as the salt when creating the key.

slen

The number of bytes in the given salt.

key

An array where the function can store the generated key. This must be FS_CRYPT0_MAX_KEY_SIZE bytes long.

klen

The number of bytes in the *key* array.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_key_gen()* function generates an encryption key from a secret password and public salt data. It uses a private implementation of the PKCS #5 PBKDF2 algorithm; you can use a different implementation if your system needs greater security.

Returns:**EOK**

Success.

EINVAL

Invalid arguments, or an error occurred in generating the key.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_migrate_control()

Control encryption migration within the filesystem

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_migrate_control( const char *path,
                              uint32_t operation,
                              uint64_t context,
                              int *preply );
```

Arguments:

path

The path to the volume on which the migration is to occur.

operation

One of the following:

- **FS_CRYPT0_MIGRATE_DISABLE** — pause and then stop the migration service.
- **FS_CRYPT0_MIGRATE_PAUSE** — pause the migration service after completing either the current work unit or the current scan for files to migrate.
- **FS_CRYPT0_MIGRATE_PERIOD** — set the period between units of migration work to be the number of milliseconds specified by the *context* argument.
- **FS_CRYPT0_MIGRATE_START** — initiate migration of files within the volume. Specific files and/or quantity of work are determined internally unless overridden by a different migration control.
- **FS_CRYPT0_MIGRATE_STOP** — stop migration work after the current work unit is completed.
- **FS_CRYPT0_MIGRATE_UNITS** — set the number of filesystem blocks to migrate in one work unit to be the value of the *context* argument.

context

Specific data for the migration command.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the `-l fscrypto` option to `qcc` to link against this library.

Description:

The `fs_crypto_migrate_control()` function controls the operation and the characteristics of the encryption migration within the filesystem.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_ALREADY

The migration service is already in the requested state.

FS_CRYPTO_REPLY_COMPLETE

The operation has been successfully completed.

FS_CRYPTO_REPLY_ERRNO

See the error code for more information.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_NOSUPPORT

The specified operation is invalid.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

ENOTSUP

The specified operation is invalid.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403), [open\(\)](#) (p. 1905), or [pthread_mutex_lock\(\)](#) (p. 2318).

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_migrate_status()

Get the status of migration in the filesystem

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_migrate_status( const char *path,
                             migrate_status_t *status,
                             int *preply );
```

Arguments:

path

The path to the filesystem's mountpoint.

status

A pointer to a `migrate_status_t` structure where the function can store the status information; see below.

preply

A pointer to a location where the function can store additional success or error information.

Library:

`libfscrypto`

Use the `-l fscrypto` option to `qcc` to link against this library.

Description:

The `fs_crypto_migrate_status()` function queries the status of migration within the filesystem.

The `migrate_status_t` structure is defined as follows:

```
typedef struct migrate_status_s {
    uint32_t      s_version;           /* The version of this structure. */
    migrate_state_t s_state;           /* The current state of migration. */
    uint64_t      s_remaining;        /* Number of bytes to be migrated. */
    uint64_t      s_migrate_inode;    /* Inode # of last migration unit. */
    int           s_migrate_error;    /* Error of last migration unit. */
    /* Version 1 END */
} migrate_status_t;
```

You must specify the version of the status information in the provided status structure's `s_version` field. The status provided will match that of the version specified. Any fields in the structure that are not valid for the version provided will contain zeroed data and are invalid.

You can inject an error into the migration process by specifying an error value in the `s_migrate_error` field of the status structure. If this field contains any value other than `EOK` (0), that error is propagated into the migration process, detected on the next state machine transition. The error won't be reported by `fs_crypto_migrate_status()` until the next state machine transition occurs. If no error is to be injected, set the `s_migrate_error` field to `EOK`.

This function sets the variable pointed to by `preply` to one of the following values:

FS_CRYPTO_REPLY_COMPLETE

The status information was successfully obtained.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

Returns:

EOK

Success.

EINVAL

Invalid arguments.

ENOMEM

Insufficient free memory.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_migrate_path()

Mark an entire directory for migration into an encryption domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_migrate_path( const char *path,
                           int domain,
                           int *preply );
```

Arguments:

path

The path to the directory that you want to migrate.

domain

The domain that you want the directory to migrate to. This can be 0 if you want it to move to the unencrypted domain.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_migrate_path()* function marks the given directory and its contents for migration to the given domain.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_BADF

The path specifies a restricted file that can't migrate.

FS_CRYPTO_REPLY_COMPLETE

The directory was successfully marked.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_READONLY

The filesystem is mounted read-only.

FS_CRYPTO_REPLY_NOSUPPORT

The file isn't a regular file.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

This function can also return any of the errors indicated by [calloc\(\)](#) (p. 255), [devctl\(\)](#) (p. 403), [lstat64\(\)](#) (p. 1558), [open\(\)](#) (p. 1905), or [opendir\(\)](#) (p. 1913).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_migrate_tag()

Mark a file for migration into an encryption domain

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_migrate_tag( const char *path,
                           int domain,
                           int *preply );
```

Arguments:

path

The path of the file to mark for migration.

domain

The domain that the file is to migrate to. This can be 0 if you want the file to move to the unencrypted domain.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_migrate_tag()* function marks a file for migration into an encryption domain. When you've marked all the files that you want to migrate, call [*fs_crypto_migrate_control\(\)*](#) (p. 757) to start the background encryption.

For directories, this operation is functionally equivalent to [*fs_crypto_file_set_domain\(\)*](#) (p. 766). Regular files may have unencrypted data contents.

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_BADF

The path specifies a restricted file that can't migrate.

FS_CRYPTO_REPLY_COMPLETE

The file was successfully marked.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_READONLY

The filesystem is mounted read-only.

FS_CRYPTO_REPLY_NOSUPPORT

The file isn't a regular file.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

ENOTSUP

The file isn't a regular file.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fs_crypto_file_set_domain()

Set the domain of a file or directory

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

int fs_crypto_file_set_domain( const char *path,
                              int pdomain,
                              int *preply );
```

Arguments:

path

The path to the file or directory whose domain you want to set.

domain

The number of the domain that you want to assign the file or directory to, or 0 (FS_CRYPTO_UNASSIGNED_DOMAIN) if you don't want the item to be in any domain, and hence unencrypted.

preply

A pointer to a location where the function can store additional success or error information.

Library:

libfscrypto

Use the -l fscrypto option to gcc to link against this library.

Description:

The *fs_crypto_file_set_domain()* function sets the domain number for the given file or directory, storing the number in the location that *domain* points to.

In order for you to successfully set the domain:

- the domain must exist
- the path must be either a directory or a regular file
- there must be no data in the file
- the disk must not be read-only

This function sets the variable pointed to by *preply* to one of the following values:

FS_CRYPTO_REPLY_ALREADY

The file or directory is already assigned to the given domain.

FS_CRYPTO_REPLY_BAD_FILE

The path isn't for a directory or a regular file.

FS_CRYPTO_REPLY_COMPLETE

The function retrieved the domain number.

FS_CRYPTO_REPLY_ERRNO

See the error code for more information.

FS_CRYPTO_REPLY_INVALID

The command wasn't completed successfully.

FS_CRYPTO_REPLY_NOT_ZERO

The path is for a regular file, and the file's size isn't zero.

FS_CRYPTO_REPLY_READONLY

The filesystem is mounted as read-only.

FS_CRYPTO_REPLY_UNKNOWN_DOMAIN

There's no domain with the given number.

Returns:**EOK**

Success.

EINVAL

Invalid arguments.

This function can also return any of the errors indicated by [devctl\(\)](#) (p. 403) or [open\(\)](#) (p. 1905), or any [errno](#) (p. 539) values returned by the underlying filesystem.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

fs_crypto_set_logging()

Set the logging destination and verbosity

Synopsis:

```
#include <fs_crypto_api.h>
#include <sys/fs_crypto.h>

void fs_crypto_set_logging( FILE *logfile,
                           const int verbosity );
```

Arguments:

logfile

The file (e.g. *stdout*) that you want to send log messages to, or NULL if you want to disable logging.

verbosity

The level of verbosity for the logs, or -1 to disable logging.

Library:

libfscrypto

Use the -l fscrypto option to qcc to link against this library.

Description:

The *fs_crypto_set_logging()* function sets the logging destination and verbosity. By default, logging is disabled.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

fscanf()

Scan input from a stream

Synopsis:

```
#include <stdio.h>

int fscanf( FILE* fp,
            const char* format,
            ... );
```

Arguments:

fp

The stream that you want to read from.

format

A string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fscanf()* function scans input from the stream specified by *fp*, under control of the argument *format*.



The *fscanf()* function is generally considered unsafe for string handling; it's safer to use [fgets\(\)](#) (p. 652) to get a line of input and then use [sscanf\(\)](#) (p. 3162) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values ([errno](#) (p. 539) is set).

Examples:

Scan a date in the form "Friday March 26 1999":

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int day;
    int year;
    char weekday[10];
    char month[10];
    FILE *in_data;

    in_data = fopen( "file", "r" );
    if( in_data != NULL ) {
        fscanf( in_data, "%10s %10s %d %d",
                weekday, month, &day, &year );

        printf( "Weekday=%s Month=%s Day=%d Year=%d\n",
                weekday, month, day, year );

        fclose( in_data );

        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fseek(), fseeko(), fseeko64()

Change the current position of a stream

Synopsis:

```
#include <stdio.h>

int fseek( FILE *fp,
           long offset,
           int whence );

int fseeko( FILE *fp,
           off_t offset,
           int whence );

int fseeko64( FILE *fp,
             off64_t offset,
             int whence );
```

Arguments:

fp

A FILE pointer returned by [fopen\(\)](#) (p. 681) or [freopen\(\)](#) (p. 721).

offset

The position to seek to, relative to one of the positions specified by *whence*.

whence

The position from which to apply the offset; one of:

SEEK_SET

Compute the new file position relative to the start of the file. The value of *offset* must not be negative.

SEEK_CUR

Compute the new file position relative to the current file position. The value of *offset* may be positive, negative or zero. A SEEK_CUR with a 0 *offset* is necessary when you want to switch from reading to writing on a stream opened for updates.

SEEK_END

Compute the new file position relative to the end of the file.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `fseek()`, `fseeko()`, and `fseeko64()` functions change the current position of the stream specified by `fp`. This position defines the character that will be read or written by the next I/O operation on the file. The difference between these functions is the data type of the `offset` argument.

These functions clear the end-of-file indicator, and undo any effects of the `ungetc()` (p. 3573) function on the stream.

You can use `ftell()`, `ftello()`, or `ftello64()` (p. 789) to get the current position of the stream before changing it. You can restore the position by using the value returned by one of the `ftell()` functions in a subsequent call to the corresponding `fseek()` function with the `whence` parameter set to `SEEK_SET`.

Returns:

0

Success

-1

An error occurred; `errno` (p. 539) is set.

Errors:

These functions fail if, either the stream is unbuffered or the stream's buffer needed to be flushed, and the call to `fseek()`, `fseeko()`, or `fseeko64()` causes an underlying `lseek()` (p. 1555) or `write()` (p. 3737) to be invoked, and:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor underlying the stream file isn't open for writing, or the stream's buffer needed to be flushed and the file isn't open.

EFBIG

One of the following:

- An attempt was made to write a file that exceeds the maximum file size.
- An attempt was made to write a file that exceeds the process's file size limit.
- The file is a regular file, and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EINTR

The write operation was terminated due to the receipt of a signal; no data was transferred.

EINVAL

The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.

EIO

One of the following:

- A physical I/O error has occurred.
- The process is a member of a background process group attempting to perform a *write()* to its controlling terminal, *TOSTOP* is set, the process is neither ignoring nor blocking *SIGTTOU*, and the process group of the process is orphaned.
- (QNX Neutrino extension) The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOSPC

There was no free space remaining on the device containing the file.

ENXIO

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EOVERFLOW

- For *fseek()*, the resulting file offset would be a value that can't be represented correctly in an object of type `long`.
- For *fseeko()*, the resulting file offset would be a value that can't be represented correctly in an object of type `off_t`.

EPIPE

An attempt was made to write to a pipe or FIFO that isn't open for reading by any process; a *SIGPIPE* signal is also sent to the thread.

ESPIPE

The file descriptor underlying stream is associated with a pipe or FIFO.

ENOSYS

The underlying device is incapable of seeking.

Examples:

Determine the size of a file, by saving and restoring the current position of the file:

```
#include <stdio.h>
#include <stdlib.h>

long filesize( FILE *fp )
{
    long int save_pos;
    long size_of_file;

    /* Save the current position. */
    save_pos = ftell( fp );

    /* Jump to the end of the file. */
    fseek( fp, 0L, SEEK_END );

    /* Get the end position. */
    size_of_file = ftell( fp );

    /* Jump back to the original position. */
    fseek( fp, save_pos, SEEK_SET );

    return( size_of_file );
}

int main( void )
{
    FILE *fp;

    fp = fopen( "file", "r" );

    if( fp != NULL ) {
        printf( "File size=%ld\n", filesize( fp ) );
        fclose( fp );
    }

    return EXIT_SUCCESS;
}

return EXIT_FAILURE;
```

Classification:

fseek() is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *fseeko()* is [POSIX 1003.1](#) (p. 46); *fseeko64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fsetpos()

Set the current position of a file

Synopsis:

```
#include <stdio.h>

int fsetpos( FILE* fp,
             const fpos_t* pos );
```

Arguments:

fp

The stream whose position you want to set.

pos

A pointer to a `fpos_t` object that specifies the new position for the stream. You must have initialized the value pointed to by *pos* by calling [fgetpos\(\)](#) (p. 650) on the same file.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fsetpos()* function sets the current position of the stream specified by *fp* according to the value of the `fpos_t` object pointed to by *pos*.

Returns:

0 for success, or nonzero if an error occurs ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor underlying the stream file isn't open for writing, or the stream's buffer needed to be flushed and the file isn't open.

EFBIG

One of the following:

- An attempt was made to write a file that exceeds the maximum file size.
- An attempt was made to write a file that exceeds the process's file size limit.
- The file is a regular file, and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EINTR

The write operation was terminated due to the receipt of a signal; no data was transferred.

EINVAL

The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.

EIO

One of the following:

- A physical I/O error has occurred.
- The process is a member of a background process group attempting to perform a *write()* to its controlling terminal, *TOSTOP* is set, the process is neither ignoring nor blocking *SIGTTOU*, and the process group of the process is orphaned.
- (QNX Neutrino extension) The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOSPC

There was no free space remaining on the device containing the file.

ENXIO

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

ESPIPE

The file descriptor underlying stream is associated with a pipe or FIFO.

ENOSYS

The underlying device is incapable of seeking.

Examples:

See [fgetpos\(\)](#) (p. 650).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fstat(), fstat64()

Get file information, given a file description

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>

int fstat( int filedes,
           struct stat* buf );

int fstat64( int filedes,
             struct stat64* buf );
```

Arguments:***filedes***

The descriptor of the file that you want to get information about.

buf

A pointer to a buffer where the function can store the information about the file.

Library:**libc**

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `fstat()` and `fstat64()` functions get information from the file specified by *filedes* and stores it in the structure pointed to by *buf*.

The file `<sys/stat.h>` contains definitions for `struct stat`, as well as following macros:

S_ISBLK(m)

Test for block special file.

S_ISCHR(m)

Test for character special file.

S_ISDIR(*m*)

Test for directory.

S_ISFIFO(*m*)

Test for FIFO.

S_ISLNK(*m*)

Test for symbolic link.

S_ISREG(*m*)

Test for regular file.

S_TYPEISMQ(*buf*)

Test for message queue.

S_TYPEISSEM(*buf*)

Test for semaphore.

S_TYPEISSHM(*buf*)

Test for shared memory object.

The arguments to the macros are:

m

The value of *st_mode* in a *stat* structure.

buf

A pointer to a *stat* structure.

The macros evaluate to nonzero if the test is true, and zero if the test is false.

Access permissions are specified as a combination of bits in the *st_mode* field of the *stat* structure. These bits are defined in `<sys/stat.h>`. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

The *st_mode* field also encodes the following bits:

S_ISUID

Set user ID on execution. The process's effective user ID (EUID) is set to that of the owner of the file when the file is run as a program. On a regular file, this bit may be cleared for security reasons on any write.

S_ISGID

Set group ID on execution. Set effective group ID (EGID) on the process to the file's group when the file is run as a program. On a regular file, this bit may be cleared for security reasons on any write.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The *filedes* argument isn't a valid file descriptor.

ENOSYS

The *fstat()* function isn't implemented for the filesystem specified by *filedes*.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number can't be represented correctly in the structure pointed to by *buf*.

Examples:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes;
    int rc;
    struct stat buf;

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        rc = fstat( filedes, &buf );
        if( rc != -1 ) {
            printf( "File size = %d\n", buf.st_size );
        }

        close( filedes );
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

fstat() is [POSIX 1003.1](#) (p. 46); *fstat64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fstatvfs()*, *fstatvfs64()

Get filesystem information, given a file descriptor

Synopsis:

```
#include <sys/statvfs.h>

int fstatvfs( int fildes,
              struct statvfs *buf );

int fstatvfs64( int fildes,
                struct statvfs64 *buf );
```

Arguments:

fildes

The descriptor for a file that resides on the filesystem that you want to get information about.

buf

A pointer to a buffer where the function can store information about the filesystem; see below.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *fstatvfs()* and *fstatvfs64()* functions return a “generic superblock” describing a filesystem that you can use to get information about mounted filesystems. The *fstatvfs64()* function is a 64-bit version of *fstatvfs()*. The [statvfs\(\)](#) and [statvfs64\(\)](#) (p. 3170) functions are similar, but they take a path name instead of a file descriptor.

The *fildes* argument is an open file descriptor, obtained from a successful call to [open\(\)](#) (p. 1905), [creat\(\)](#) (p. 381), [dup\(\)](#) (p. 483), [fcntl\(\)](#) (p. 623), or [pipe\(\)](#) (p. 2011), for a file that resides on that filesystem. The filesystem type is known to the operating system. Read, write, or execute permission for the named file isn't required.

The *buf* argument is a pointer to a `statvfs` or `statvfs64` structure that's filled by the function. It contains at least:

unsigned long *f_bsize*

The preferred filesystem blocksize.

unsigned long *f_fsize*

The fundamental filesystem blocksize (if supported)

fsblkcnt_t *f_blocks*

The total number of blocks on the filesystem, in units of *f_fsize*.

fsblkcnt_t *f_bfree*

The total number of free blocks.

fsblkcnt_t *f_bavail*

The number of free blocks available to a nonsuperuser.

fsfilcnt_t *f_files*

The total number of file nodes (inodes).

fsfilcnt_t *f_ffree*

The total number of free file nodes.

fsfilcnt_t *f_favail*

The number of inodes available to a nonsuperuser.

unsigned long *f_fsid*

The filesystem ID (currently the device ID).

char *f_basetype*[16]

The type of the target filesystem, as a null-terminated string.

unsigned long *f_flag*

A bitmask of flags; the function can set these flags (the `_MOUNT_*` and `ST_*` bits are equivalent):

<code>_MOUNT_*</code>	<code>ST_*</code>	Meaning
<code>_MOUNT_RDONLY</code>	<code>ST_RDONLY</code>	The filesystem is read-only.
<code>_MOUNT_NOEXEC</code>	<code>ST_NOEXEC</code>	The filesystem doesn't permit the loading of executables.

_MOUNT_*	ST_*	Meaning
_MOUNT_NOSUID	ST_NOSUID	The filesystem doesn't support setuid() (p. 2917) and setgid() (p. 2858) semantics.
_MOUNT_NOCREAT	ST_NOCREAT	You can't create files on the filesystem.
_MOUNT_OFF32	ST_OFF32	The <code>off_t</code> type is limited to 32 bits.
_MOUNT_NOATIME	ST_NOATIME	The filesystem doesn't support the logging of file access times.

unsigned long *f_namemax*

The maximum filename length.

The values returned for *f_files*, *f_ffree*, and *f_favail* depend on the filesystem:

fs-dos.so, fs-qnx4.so

These filesystems embed inodes in directories, so the number of files is limited by the amount of disk space, so basically there's no limit. They don't count the number of files created either, so they set all these fields to 0.

fs-qnx6.so, fs-etfs-ram, fs-ext2.so

These filesystems have a fixed inodes table, so they fill in these fields.

fs-cd.so, fs-mac.so, fs-nt.so, fs-udf.so

Read-only fsystems always set *f_ffree* and *f_favail* to 0. Where possible or known, they try to set *f_files*:

- *fs-mac.so* knows exactly
- *fs-nt.so* provides an upper bound based on the current size of the Master File Table (MFT)
- *fs-udf.so* knows when it's a UDF that's mounted; ISO formats don't indicate that information, so *f_files* is set to 0
- *fs-cd.so* sets it to 0

fs-cifs

The CIFS filesystem sets these fields to 0.

devf - *

Flash filesystem drivers estimate the values of these fields, based on the amount of free space.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The *fildev* argument isn't an open file descriptor.

EFAULT

The *buf* argument points to an illegal address.

EINTR

A signal was caught during execution.

EIO

An I/O error occurred while reading the filesystem.

EOVERFLOW

One of the values to be returned can't be represented correctly in the structure pointed to by *buf*.

Classification:

fstatvfs() is [POSIX 1003.1 XSI](#) (p. 46); *fstatvfs64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fsync()

Synchronize the file state

Synopsis:

```
#include <unistd.h>

int fsync( int filedes );
```

Arguments:

filedes

The descriptor for the file that you want to synchronize.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fsync()* function forces all queued I/O operations for the file specified by the *filedes* file descriptor to finish, synchronizing the file's state. The function blocks until this is finished. For more information about synchronizing, see “Filesystems and block I/O (devb - *) drivers” in the Fine-Tuning Your System chapter of the QNX Neutrino *User's Guide*.

Although similar to [fdatsync\(\)](#) (p. 631), *fsync()* also guarantees the integrity of file information such as access and modification times.

Returns:

0 for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

The *filedes* argument isn't a valid file descriptor open for writing.

EINVAL

The implementation doesn't support synchronized I/O for the given file.

ENOSYS

The *fsync()* function isn't supported for the filesystem specified by *filedes*.

Classification:

POSIX 1003.1 FSC (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ftell(), ftello(), ftello64()

Return the current position of a stream

Synopsis:

```
#include <stdio.h>

long int ftell( FILE* fp );
off_t ftello( FILE* fp );
off64_t ftello64( FILE* fp );
```

Arguments:

fp

The stream that you want to get the current position of.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ftell()`, `ftello()`, and `ftello64()` functions return the current position of the stream specified by `fp`. This position defines the character that will be read or written by the next I/O operation on the file. The difference between these functions is the data type of the returned position.

You can use the value returned by an `ftell()` function in a subsequent call to [fseek\(\)](#), [fseeko\(\)](#), or [fseeko64\(\)](#) (p. 772) to restore the file position to a previous value.

Returns:

The current position of the file or `-1L` if an error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

long filesize( FILE *fp )
{
    long int save_pos;
    long size_of_file;

    /* Save the current position. */
```

```
    save_pos = ftell( fp );

    /* Jump to the end of the file. */
    fseek( fp, 0L, SEEK_END );

    /* Get the end position. */
    size_of_file = ftell( fp );

    /* Jump back to the original position. */
    fseek( fp, save_pos, SEEK_SET );

    return( size_of_file );
}

int main( void )
{
    FILE *fp;

    fp = fopen( "file", "r" );

    if( fp != NULL ) {
        printf( "File size=%ld\n", filesize( fp ) );
        fclose( fp );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

ftell() is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *ftello()* is [POSIX 1003.1](#) (p. 46);
ftello64() is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

ftime()

Get the current time

Synopsis:

```
#include <sys/timeb.h>

int ftime( struct timeb * timeptr );
```

Arguments:

timeptr

A pointer to a `timeb` structure where the function can store the current time; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ftime()` function stores the current time in the *timeptr* structure. The `timeb` structure contains the following fields:

`time_t` *time*

Time, in seconds, since the Unix Epoch, 00:00:00 January 1, 1970 Coordinated Universal Time (UTC).

`unsigned short` *millitm*

Milliseconds.

`short` *timezone*

Difference in minutes of the timezone from UTC.

`short` *dstflag*

Nonzero if in daylight savings time.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).**Examples:**

```
#include <stdio.h>
#include <time.h>
#include <sys/timeb.h>
#include <stdlib.h>

int main( void )
{
    struct timeb timebuf;
    char *now;

    ftime( &timebuf );
    now = ctime( &timebuf.time );

    /* Note that we're cutting "now" off
     * after 19 characters to avoid the
     * \n that ctime() appends to the
     * formatted time string.
     */

    printf( "The time is %.19s.%hu\n",
           now, timebuf.millitm );

    return EXIT_SUCCESS;
}
```

Produces output similar to the following:

```
The time is Mon Jul 05 15:58:42.870
```

Classification:[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ftruncate()*, *ftruncate64()

Truncate a file

Synopsis:

```
#include <unistd.h>

int ftruncate( int fildes,
               off_t length );

int ftruncate64( int fildes,
                 off64_t length );
```

Arguments:

fildes

The descriptor for the file that you want to truncate.

length

The length that you want the file to be, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ftruncate()* and *ftruncate64()* functions cause the file referenced by *fildes* to have a size of *length* bytes. The *ftruncate64()* function is a 64-bit version of *ftruncate()*.

If the size of the file previously exceeded *length*, the extra data is discarded (this is similar to using the `F_FREESP` option with *fcntl()* (p. 623)). If the size of the file was previously shorter than *length*, the file size is extended with NUL characters (similar to the `F_ALLOCS` option to *fcntl()*).

These functions don't modify the value of the seek pointer.

Upon successful completion, the *ftruncate()* function marks the *st_ctime* and *st_mtime* fields of the file for update. If the *ftruncate()* function is unsuccessful, the file is unaffected.

Returns:

Zero for success, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

The *fil-des* argument isn't a valid file descriptor.

EFBIG

The file is a regular file and *length* is greater than the offset maximum associated with the file.

EINTR

A signal was caught during the call to *ftruncate()*.

EINVAL

The *fil-des* argument doesn't refer to a file on which this operation is possible, the *file-des* argument isn't open for writing or the *length* argument is less than the minimum file size for the specified filesystem.

EIO

An I/O error occurred while reading from or writing to the filesystem.

ENOMEM

There wasn't enough memory to change the size of a shared memory object.

ENOSYS

The *ftruncate()* function isn't implemented for the filesystem specified by *file-des*.

ENOTSUP

The *ftruncate()* function is implemented for the specified filesystem, but the specific operation (F_ALLOCSP or F_FREESP; see [fcntl\(\)](#) (p. 623)) isn't supported.

EROFS

The file resides on a read-only filesystem.

Classification:

ftruncate() is [POSIX 1003.1](#) (p. 46); *ftruncate64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ftrylockfile()

Acquire ownership of a file, without blocking

Synopsis:

```
#include <stdio.h>

int ftrylockfile( FILE* file );
```

Arguments:

file

A pointer to the FILE object for the file you want to lock.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *ftrylockfile()* function is used by a thread to acquire ownership of a FILE if the object is available; *ftrylockfile()* is a nonblocking version of *flockfile()* (p. 669). To unlock the file, call *funlockfile()* (p. 801).

Returns:

0

Success.

Nonzero

The lock can't be acquired.

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

ftw()*, *ftw64()

Walk a file tree

Synopsis:

```
#include <ftw.h>

int ftw( const char *path,
         int (*fn)( const char *fname,
                    const struct stat *sbuf,
                    int flags),
         int depth );
```

Arguments:

path

The path of the directory whose file tree you want to walk.

fn

A pointer to a function that you want to call for each file; see below.

depth

The maximum number of file descriptors that *ftw()* can use. The *ftw()* function uses one file descriptor for each level in the tree.

If *depth* is zero or negative, the effect is the same as if it were 1. The *depth* must not be greater than the number of file descriptors currently available for use. The *ftw()* function is faster if *depth* is at least as large as the number of levels in the tree.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



These functions are in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *ftw()* and *ftw64()* functions recursively descend the directory hierarchy identified by *path*. The *ftw64()* function is a 64-bit version of *ftw()*.

For each object in the hierarchy, *ftw()* calls the user-defined function *fn()*, passing to it:

- a pointer to a NULL-terminated character string containing the name of the object
- a pointer to a `stat` structure (see [stat\(\)](#) (p. 3164)) containing information about the object
- an integer. Possible values of the integer, defined in the `<ftw.h>` header, are:

FTW_F

The object is a file.

FTW_D

The object is a directory.

FTW_DNR

The object is a directory that can't be read. Descendants of the directory aren't processed.

FTW_NS

The *stat()* failed on the object because the permissions weren't appropriate, or the object is a symbolic link that points to a nonexistent file. The *stat* buffer passed to *fn()* is undefined.

The *ftw()* function visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn()* returns a nonzero value, or some error is detected within *ftw()* (such as an I/O error). If the tree is exhausted, *ftw()* returns zero. If *fn()* returns a nonzero value, *ftw()* stops its tree traversal and returns whatever value was returned by *fn()*.

When *ftw()* returns, it closes any file descriptors it opened; it doesn't close any file descriptors that may have been opened by *fn()*.

Returns:

0

Success.

-1

An error (other than EACCES) occurred ([errno](#) (p. 539) is set).

Classification:

ftw() is [POSIX 1003.1 XSI](#) (p. 46); *ftw64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Because *ftw()* is recursive, it might terminate with a memory fault when applied to very deep file structures.

This function uses *malloc()* to allocate dynamic storage during its operation. If *ftw()* is forcibly terminated, for example if *longjmp()* is executed by *fn()* or an interrupt routine, *ftw()* doesn't have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn()* return a nonzero value at its next invocation.

funlockfile()*Release ownership of a file***Synopsis:**

```
#include <stdio.h>

void funlockfile( FILE* file );
```

Arguments:***file***

A pointer to the FILE object for the file you want to unlock.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *funlockfile()* function is used to release ownership of *file* granted to the thread by a call to *flockfile()* (p. 669) or *ftrylockfile()* (p. 796). The behavior is undefined if a thread other than the current owner calls the *funlockfile()* function.

For more information, see *flockfile()* (p. 669).

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

futime()

Record the modification time for a file

Synopsis:

```
#include <utime.h>

int futime( int fildes,
            const struct utimbuf *times );

struct utimbuf {
    time_t    actime;    /* access time */
    time_t    modtime;   /* modification time */
};
```

Arguments:

fildes

The descriptor for the file whose modification time you want to get or set.

times

NULL, or a pointer to a `utimbuf` structure where the function can store the modification time.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `futime()` function records the modification time for the file or directory with the descriptor, *fildes*.

If the *times* argument is NULL, the access and modification times of the file or directory are set to the current time. The effective user ID of the process must match the owner of the file or directory, or the process must have write permission to the file or directory, or appropriate privileges in order to use the `futime()` function in this way.

If the *times* argument isn't NULL, it's interpreted as a pointer to a `utimbuf` structure, and the access and modification times of the file or directory are set to the values contained in the *actime* and *modtime* fields in this structure. Only the owner of the file or directory, and processes with appropriate privileges are permitted to use the `futime()` function in this way.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539)) is set.**Errors:****EACCES**

Search permission is denied for a component of *path*, or the *times* argument is NULL, and the effective user ID of the process doesn't match the owner of the file, and write access is denied.

ENAMETOOLONG

The argument *path* exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.

ENOENT

The specified *path* doesn't exist, or *path* is an empty string.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The *times* argument isn't NULL, and the calling process's effective user ID has write access to the file but doesn't match the owner of the file, and the calling process doesn't have the appropriate privileges.

EROFS

The named file resides on a read-only filesystem.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fwide()

Set or get the stream orientation

Synopsis:

```
#include <wchar.h>

int fwide( FILE * fp,
           int mode );
```

Arguments:

fp

The stream whose orientation you want to set.

mode

The orientation mode:

- If *mode* is greater than zero and the stream orientation hasn't been set, *fwide()* flags the stream as wide-oriented.
- If *mode* is less than zero, *fwide()* behaves similarly, but flags the stream as byte-oriented.
- If *mode* is zero, *fwide()* returns the stream type without altering the stream.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fwide()* function sets or determines the orientation of the stream *fp*.

Returns:

> 0

The stream is (now) wide-oriented.

0

The stream is unbound.

< 0

The stream is (now) byte-oriented.

Errors:

EBADF

The *fp* argument isn't valid.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

fwprintf()

Write wide-character output to a stream

Synopsis:

```
#include <wchar.h>

int fwprintf( FILE * fp,
              const wchar_t * format,
              ... );
```

Arguments:

fp

The stream to which you want to send the output.

format

A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fwprintf()* function writes output to the stream specified by *fp*, under control of the *format* specifier.

The *fwprintf()* function is the wide-character version of [fprintf\(\)](#) (p. 702).

Returns:

The number of wide characters written, excluding the terminating NUL, or a negative number if an error occurred ([errno](#) (p. 539) is set).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fwrite()

Write elements to a file

Synopsis:

```
#include <stdio.h>

size_t fwrite( const void* buf,
               size_t size,
               size_t num,
               FILE* fp );
```

Arguments:

buf

A pointer to a buffer that contains the elements that you want to write.

size

The size of each element to write.

num

The number of elements to write.

fp

The stream to which to write the elements.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fwrite()* function writes *num* elements of *size* bytes each to the stream specified by *fp*.

Returns:

The number of complete elements successfully written; if an error occurs, this is less than *num*.

Errors:

If an error occurs, [errno](#) (p. 539) is set to indicate the type of error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

struct student_data {
    int student_id;
    unsigned char marks[10];
};

int main( void )
{
    FILE *fp;
    struct student_data std;
    int i;

    fp = fopen( "file", "w" );
    if( fp != NULL ) {
        std.student_id = 1001;

        for( i = 0; i < 10; i++ ) {
            std.marks[i] = (unsigned char)(85 + i);
        }

        /* write student record with marks */
        i = fwrite( &std, sizeof( struct student_data ), 1, fp );
        printf( "Successfully wrote %d records\n", i );

        fclose( fp );

        if( i == 1 ) {
            return EXIT_SUCCESS;
        }
    }

    return EXIT_FAILURE;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

fwscanf()

Scan wide-character input from a stream

Synopsis:

```
#include <wchar.h>

int fwscanf( FILE * fp,
             const wchar_t * format,
             ... );
```

Arguments:

fp

The stream that you want to read from.

format

A wide-character string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *fwscanf()* function scans input from the stream specified by *fp*, under control of the argument *format*. Following the format string is a list of addresses to receive values.

The *fwscanf()* function is the wide-character version of [fscanf\(\)](#) (p. 770).



The *fwscanf()* function is generally considered unsafe for string handling; it's safer to use [fgetws\(\)](#) (p. 658) to get a line of input and then use [swscanf\(\)](#) (p. 3274) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 9

G

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>get_device_command()</i>	<i>devctl()</i> (p. 403)
<i>get_device_direction()</i>	<i>devctl()</i> (p. 403)

The following have been deprecated:

Instead of:	Use:
<i>getpriority()</i>	<i>pthread_getschedparam()</i> (p. 2299) or <i>SchedGet()</i> (p. 2774)

Note the following:

- [*getprio\(\)*](#) (p. 922) is for compatibility with QNX 4 programs; use [*pthread_getschedparam\(\)*](#) (p. 2299) or [*SchedGet\(\)*](#) (p. 2774)

This chapter describes the API components whose names begin with “G”.

gai_strerror()

Return the string associated with a [getaddrinfo\(\)](#) error code

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

const char * gai_strerror( int ecode );
```

Arguments:

ecode

The error code number from the [getaddrinfo\(\)](#) (p. 818) function.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *gai_strerror()* function returns a string describing the error code from the [getaddrinfo\(\)](#) (p. 818) function. Nonzero error codes are defined in `<netdb.h>` as follows:

EAI_ADDRFAMILY

The address family for *nodename* isn't supported.

EAI_AGAIN

There was a temporary failure in name resolution.

EAI_BADFLAGS

Invalid value for *ai_flags*.

EAI_FAIL

Nonrecoverable failure in name resolution.

EAI_FAMILY

The *ai_family* isn't supported.

EAI_MEMORY

Memory allocation failure.

EAI_NODATA

No address associated with the *nodename*.

EAI_NONAME

Either the *nodename* or the *servname* argument wasn't provided or isn't known.

EAI_SERVICE

The *servname* argument isn't supported for *ai_socktype*.

EAI_SOCKTYPE

The *ai_socktype* isn't supported.

EAI_SYSTEM

System error returned in [errno](#) (p. 539).

Returns:

If called with a proper *ecode* argument, a pointer to a string describing the given error code. If the argument isn't one of the `EAI_*` values, a pointer to a string whose contents indicate an unknown error.



Don't modify the strings that this function returns.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

gamma()*, *gamma_r()*, *gammaf()*, *gammaf_r()

Log gamma function

Synopsis:

```
#include <math.h>

double gamma( double x );
double gamma_r( double x,
                int* signgam);
float gammaf( float x );
float gammaf_r( float x,
                int* signgam);
```

Arguments:

x

An arbitrary number.

signgam

(*gamma_r()*, *gammaf_r()* only) A pointer to a location where the function can store the sign of *x*.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *gamma()* and *gamma_r()* functions return the natural log (ln) of the *gamma()* function and are equivalent to [lgamma\(\)](#) (p. 1513). These functions return $\ln |\Gamma(x)|$, where $\Gamma(x)$ is defined as follows:

For $x > 0$:

$$\int_0^{\infty} e^{-t} t^{x-1} dt$$

For $x < 1$:

$$n / ((1-x) * \sin(nx))$$

The results converge when x is between 0 and 1. The function has the property:

$$\Gamma(N) = (N-1) \times \Gamma(N-1)$$

The *gamma** functions compute the log because the function grows very quickly.

The *gamma()* and *gammaf()* functions use the external integer *signgam* to return the sign of $\Gamma(x)$, while *gamma_r()* and *gammaf_r()* use the user-allocated space addressed by *signgamp*.

The *signgam* variable isn't set until *gamma()* or *gammaf()* returns. For example, don't use the expression:



```
g = signgam * exp( gamma( x ) );
```

to compute $g = \Gamma(x)'$. Instead, compute *gamma()* first:

```
lg = gamma(x);  
g = signgam * exp( lg );
```

Note that $\Gamma(x)$ must overflow when x is large enough, underflow when $-x$ is large enough, and generate a division by 0 exception at the singularities x a nonpositive integer.

Returns:

$$\ln | \Gamma(x) |$$



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set *errno* (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

getaddrinfo()

Get socket address information

Synopsis:

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo( const char * nodename,
                 const char * servname,
                 const struct addrinfo * hints,
                 struct addrinfo ** res );
```

Arguments:

nodename

The node name. A non-NULL *nodename* may be either a node name or a numeric host address string (i.e. a dotted-decimal IPv4 address or an IPv6 hex address.)

servname

The server name. A non-NULL *servname* may be either a server name or a decimal port number.

hints

A pointer to an [addrinfo](#) (p. 122) structure that provides hints about the type of socket you're supporting. See “[Using the hints argument](#) (p. 819)” for more information.

res

The address of a location where the function can store a pointer to a linked list of one or more [addrinfo](#) structures.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getaddrinfo()* function performs the functionality of [gethostbyname\(\)](#) (p. 872) and [getservbyname\(\)](#) (p. 952) but in a more sophisticated manner.

The *nodename* and *servname* arguments are either pointers to null-terminated strings or `NULL`. One or both of these two arguments must be a non-`NULL` pointer. Normally, a client scenario specifies both *nodename* and *servname*.

On success, the *getaddrinfo()* function stores, in the location pointed to by *res*, a pointer to a linked list of one or more `addrinfo` structures. You can process each `addrinfo` structure in this list by following the *ai_next* pointer until reaching a `NULL` pointer. Each `addrinfo` structure contains the corresponding *ai_family*, *ai_socktype*, and *ai_protocol* arguments for a call to the [socket\(\)](#) (p. 3075) function. The *ai_addr* argument of the `addrinfo` structure points to a filled-in socket address structure with a length specified by the *ai_addrlen* argument.



If the name server isn't responding, there's a timeout of 1.5 minutes per name server.

Using the *hints* argument

You can optionally pass an `addrinfo` structure, pointed to by the *hints* argument, that provides hints concerning the type of socket that your application supports.

In this structure, all members — except *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* — must be zero or a `NULL` pointer. The `addrinfo` structure of the *hints* argument can accept various types of sockets:

To accept:	Set:	To:
Any protocol family	<i>ai_family</i>	<code>PF_UNSPEC</code>
Any socket type	<i>ai_socktype</i>	<code>0</code>
Any protocol	<i>ai_protocol</i>	<code>0</code>
All of the above (as well as setting <i>ai_flags</i> to <code>0</code>)	<i>hints</i>	<code>NULL</code>

The *hints* argument defaults to all possibilities, but you can also use it to limit choices:

- If the application handles only TCP but not UDP, you could set the *ai_socktype* member of the *hints* structure to `SOCK_STREAM`.
- If the application handles only IPv4 but not IPv6, you could set the *ai_family* member of the *hints* structure to `PF_INET`.

Using the *ai_flags* argument in the *hints* structure

You can set the *ai_flags* argument to further configure the *hints* structure. Settings for *ai_flags* include:

AI_PASSIVE

Set this bit if you plan to use the returned `addrinfo` structure in a call to [bind\(\)](#) (p. 208). In this call, if the *nodename* argument is a `NULL` pointer, then the IP address portion of the socket address structure *ai_addr* is set to `INADDR_ANY` for an IPv4 address or `IN6ADDR_ANY_INIT` for an IPv6 address.

If you don't set the `AI_PASSIVE` flag, you can use the returned `addrinfo` structure in a call to:

- [connect\(\)](#) (p. 348) — connectionless or connection-oriented protocol
- [sendto\(\)](#) (p. 2837) — connectionless protocol
- [sendmsg\(\)](#) (p. 2833) — connectionless protocol

In this case, if the *nodename* argument is a `NULL` pointer, then the IP address portion of the socket address structure *ai_addr* is set to the loopback address.

AI_CANONNAME

Set this bit if you want the *ai_canonname* argument of the first `addrinfo` structure to point to a null-terminated string containing the canonical name of the specified *nodename*.

AI_NUMERICHOST

Set this bit if you want to prevent any type of name resolution service (such as DNS) from being used. A non-`NULL` *nodename* string must be a numeric host address string; otherwise, *getaddrinfo()* returns `EAI_NONAME`.

Pitfalls

The arguments to *getaddrinfo()* must be sufficiently consistent and unambiguous or this function will return an error. Here are some problems you may encounter:

- Inconsistent *hints* — for Internet address families, specifying `SOCK_STREAM` for *ai_socktype* while specifying `IPPROTO_UDP` for *ai_protocol*.
- Inconsistent *servname* — specifying a *servname* that's defined only for certain *ai_socktype* values, such as the TFTP service (a datagram service `SOCK_DGRAM`) on `SOCK_STREAM`.
- Undefined service names — specifying a *servname* while specifying `SOCK_RAW` for *ai_socktype*. (Service names aren't defined for the internet `SOCK_RAW` space.)
- Incomplete specifications — specifying a numeric *servname* while leaving *ai_socktype* and *ai_protocol* unspecified. The *getaddrinfo()* function isn't allowed to [glob\(\)](#) (p. 999) the argument when a numeric *servname* doesn't have a specified socket type.

The *getaddrinfo()* function dynamically allocates space for the following:

- *addrinfo* structures
- socket address structures
- canonical node name strings pointed to by the *addrinfo* structures.



Use [freeaddrinfo\(\)](#) (p. 719) to free the *addrinfo* structures, and [gai_strerror\(\)](#) (p. 814) to decipher error codes.

Returns:

Zero for success, or nonzero if an error occurs.

Errors:

To get an explanation of any error code, use [gai_strerror\(\)](#) (p. 814).

Examples:

The following code tries to connect to `www.kame.net` service HTTP using a stream socket. It loops through all the addresses available, regardless of the address family. If the destination resolves to an IPv4 address, it uses a `AF_INET` socket. Similarly, it uses an `AF_INET6` socket if it resolves to IPv6. Note that there aren't any hardcoded references to any particular address family; the code works even if *getaddrinfo()* returns addresses that aren't IPv4/v6.

```
struct addrinfo hints, *res, *res0;
int error;
int s;
const char *cause = NULL;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
if (error) {
    err(1, "%s", gai_strerror(error));
    /*NOTREACHED*/
}
s = -1;
for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype,
               res->ai_protocol);
    if (s < 0) {
        cause = "socket";
        continue;
    }

    if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        cause = "connect";
        close(s);
        s = -1;
        continue;
    }

    break; /* okay we got one */
}
if (s < 0) {
    err(1, cause);
    /*NOTREACHED*/
}
```

```
}
freeaddrinfo(res0);
```

The following example tries to open a wildcard-listening socket onto the HTTP service for all of the available address families:

```
struct addrinfo hints, *res, *res0;
int error;
int s[MAXSOCK];
int nsock;
const char *cause = NULL;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
error = getaddrinfo(NULL, "http", &hints, &res0);
if (error) {
    err(1, "%s", gai_strerror(error));
    /*NOTREACHED*/
}
nsock = 0;
for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
    s[nsock] = socket(res->ai_family, res->ai_socktype,
        res->ai_protocol);
    if (s[nsock] < 0) {
        cause = "socket";
        continue;
    }

    if (connect(s[nsock], res->ai_addr, res->ai_addrlen) < 0) {
        cause = "connect";
        close(s[nsock]);
        continue;
    }

    nsock++;
}
if (nsock == 0) {
    err(1, cause);
    /*NOTREACHED*/
}
freeaddrinfo(res0);
```

Files:

/etc/nsswitch.conf

Name-service switch configuration file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getc()

Get the next character from a file

Synopsis:

```
#include <stdio.h>

int getc( FILE* fp );
```

Arguments:

fp

The stream you want to get the character from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getc()` macro gets the next character from the stream designated by *fp*. The character is returned as an `int` value.

Returns:

The next character from the stream *fp*, cast as `(int)(unsigned char)`, or EOF if an end-of-file or error condition occurs ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE* fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = getc( fp )) != EOF ) {
            putchar(c);
        }
    }
}
```

```
        fclose( fp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

getc() is a macro.

getc_unlocked()

Get the next character from a file

Synopsis:

```
#include <stdio.h>

int getc_unlocked( FILE *fp );
```

Arguments:

fp

The stream you want to get the character from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getc_unlocked()` function is a thread-unsafe version of [getc\(\)](#) (p. 823). You can use it safely only when the invoking thread has locked *fp* using [flockfile\(\)](#) (p. 669) (or [ftrylockfile\(\)](#) (p. 796)) and [funlockfile\(\)](#) (p. 801).

Returns:

The next character from the input stream pointed to by *fp*, or EOF if an end-of-file or error condition occurs ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

getchar()

Get a character from *stdin*

Synopsis:

```
#include <stdio.h>

int getchar( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getchar()* function is equivalent to *getc()* on the *stdin* stream.

Returns:

The next character from the input stream pointed to by *stdin*, cast as `(int)` (unsigned `char`), or EOF if an end-of-file or error condition occurs ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    /* Get characters from "file" instead of
     * stdin.
     */
    fp = freopen( "file", "r", stdin );
    while( ( c = getchar() ) != EOF ) {
        putchar(c);
    }

    fclose( fp );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getchar_unlocked()

Get a character from *stdin*

Synopsis:

```
#include <stdio.h>

int getchar_unlocked( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getchar_unlocked()` function is a thread-unsafe version of [getchar\(\)](#) (p. 827). You can use it safely only when the invoking thread has locked *stdin* using [flockfile\(\)](#) (p. 669) (or [ftrylockfile\(\)](#) (p. 796)) and [funlockfile\(\)](#) (p. 801).

Returns:

The next character from the input stream pointed to by *stdin*, cast as `(int)(unsigned char)`, or EOF if an end-of-file or error condition occurs ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getcwd()

Get the name of the current working directory

Synopsis:

```
#include <unistd.h>

char* getcwd( char* buffer,
              size_t size );
```

Arguments:

buffer

NULL, or a pointer to a buffer where the function can store the directory name.

size

The size of the buffer, in bytes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *getcwd()* function returns the name of the current working directory. *buffer* is a pointer to a buffer of at least *size* bytes where the NUL-terminated name of the current working directory will be placed.

The maximum size that might be required for *buffer* is `PATH_MAX + 1` bytes. See `<limits.h>`.



POSIX doesn't specify what should happen if you call `getcwd( NULL, 0)`.

The QNX Neutrino version (like many others) allocates a buffer for the name of the directory; it's up to your application to free the buffer when you no longer need it.

Returns:

The address of the string containing the name of the current working directory, or NULL if an error occurred (*errno* (p. 539) is set).

Errors:**EINVAL**

The argument *size* is negative or 0.

ELOOP

Too many levels of symbolic links.

ENOSYS

The *getcwd()* function isn't implemented for the filesystem specified in the current working directory.

ERANGE

The buffer is too small (as specified by *size*) to contain the name of the current working directory.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>

int main( void )
{
    char* cwd;
    char buff[PATH_MAX + 1];

    cwd = getcwd( buff, PATH_MAX + 1 );
    if( cwd != NULL ) {
        printf( "My working directory is %s.\n", cwd );
    }

    return EXIT_SUCCESS;
}
```

produces the output:

```
My working directory is /home/bill.
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

There is only one current working directory per *process*. In a multithreaded application, any thread calling [chdir\(\)](#) (p. 286) will change the current working directory for *all threads* in that process.

getdomainname()

Get the domain name of the current host

Synopsis:

```
#include <unistd.h>

int getdomainname( char * name,
                  size_t namelen );
```

Arguments:

name

A buffer where the function can store the domain name.

namelen

The size of the name array.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getdomainname()* function gets the standard domain name for the current processor and stores it in the buffer that *name* points to. The name is null-terminated.



If the buffer is too small, the name is truncated.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

The *name* or *namelen* parameters gave an invalid address.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getdtablesize()

Get the size of the file descriptor table

Synopsis:

```
#include <unistd.h>

int  getdtablesize( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The `getdtablesize()` returns the size of this table.

This function is equivalent to [getrlimit\(\)](#) (p. 942) with the `RLIMIT_NOFILE` option.

Returns:

The size of the file descriptor table.

Classification:

[Legacy Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getegid()

Get the effective group ID

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

gid_t getegid( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getegid()* function gets the effective group ID for the calling process.

Returns:

The calling process's effective group ID. This function can't fail.

Examples:

```
/*
 * Print the effective group ID of a process
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "My effective group ID is %d\n", getegid() );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

getenv()

Get the value of an environment variable

Synopsis:

```
#include <stdlib.h>

char* getenv( const char* name );
```

Arguments:

name

The name of the environment variable whose value you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getenv()* function searches the environment list for a string in the form *name=value* and returns a pointer to a string containing the *value* for the specified *name*. The matching is case-sensitive.

Returns:

A pointer to the *value* assigned to *name*, or NULL if *name* wasn't found in the environment.



Don't modify the returned string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char* path;

    path = getenv( "INCLUDE" );
    if( path != NULL ) {
        printf( "INCLUDE=%s\n", path );
        return EXIT_SUCCESS;
    }
}
```

```
    return EXIT_FAILURE;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The `getenv()` function manipulates the environment pointed to by the global [environ](#) (p. 528) variable.

geteuid()

Get the effective user ID

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

uid_t geteuid( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *geteuid()* function gets the effective user ID for the calling process.

Returns:

The calling process's effective user ID.

Examples:

```
/*
 * Print the effective user ID of a process.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "My effective user ID is %d\n", geteuid() );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

getfsfile()

Search for a filesystem name in the filesystem table (/etc/fstab) file

Synopsis:

```
#include <fstab.h>

struct fstab * getfsfile(const char *file);
```

Arguments:

file

The prefix of the filesystem that you want to search for.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getfsfile()* function searches the filesystem table (/etc/fstab) for an entry for the specified filesystem prefix.

Returns:

A pointer to the `fstab` structure for the filesystem (see the entry for *getfsent()* (p. 844)), or NULL if the entry couldn't be found.

Classification:

NetBSD (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

The functions that work with `/etc/fstab` use static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

getfsent()

Get the next entry from the filesystem table (/etc/fstab) file

Synopsis:

```
#include <fstab.h>

struct fstab * getfsent(void);
```

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getfsent()* function gets the next entry from the filesystem table file, `/etc/fstab`.

The fstab structure

The *getfsent()*, *getfsfile()* (p. 842), and *getfsspec()* (p. 847) functions return pointers to a `fstab` structure, which is defined as follows:

```
struct fstab {
    char    *fs_spec;
    char    *fs_file;
    char    *fs_vfstype;
    char    *fs_mntops;
    char    *fs_type;
    int     init_flags;
    int     init_mask;
};
```

The members include:

fs_spec

The block special device name (e.g. `/dev/hd0t177`).

fs_file

The filesystem path prefix.

fs_vfstype

The type of filesystem; for a list of types, see the entry for `mount` in the *Utilities Reference*.

fs_mntops

A comma-separated list of mount options, which can include:

String	Constant	Meaning
implied	FSTAB_IMPLIED	The root entry was implied, not specified.
allservers	FSTAB_OCB	Send the mount request to all servers.
ro	FSTAB_RO	Read-only device.
rw	FSTAB_RW	Read-write device.
xx	FSTAB_XX	Completely ignore the entry.

fs_type

The type of filesystem; one of the following:

String	Constant	Meaning
ro	FSTAB_RO	Read-only device.
rw	FSTAB_RW	Read-write device.
xx	FSTAB_XX	Completely ignore the entry.

init_flags

Flags to OR in:

- `_MFLAG_OCB` — ignore the special device string, and contact all servers.
- `_MOUNT_IMPLIED` — the mountpoint is unspecified by the client.

init_mask

Flags to AND out.

Returns:

A pointer to the `fstab` structure for the next entry in the file, or `NULL` if the function reached the end of the file.

Classification:

NetBSD (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The functions that work with `/etc/fstab` use static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

getfsspec()

Search for a block special device in the filesystem table (/etc/fstab) file

Synopsis:

```
#include <fstab.h>

struct fstab * getfsspec(const char *spec);
```

Arguments:

spec

The name of the block special device that you want to search for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getfsfile()` function searches the filesystem table (/etc/fstab) for an entry for the specified block special device.

Returns:

A pointer to the `fstab` structure for the block special device (see the entry for [getfsent\(\)](#) (p. 844)), or `NULL` if the entry couldn't be found.

Classification:

[NetBSD](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

The functions that work with `/etc/fstab` use static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

getgid()*Get the group ID***Synopsis:**

```
#include <sys/types.h>
#include <unistd.h>

gid_t getgid( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *getgid()* function gets the group ID for the calling process.

Returns:

The calling process's group ID. This function can't fail.

Examples:

```
/*
 * Print the group id of a process.
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "I belong to group ID %d\n", getgid() );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

getgrent()

Return an entry from the group database

Synopsis:

```
#include <grp.h>

struct group* getgrent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgrent()` function returns the next entry from the group database, although no particular order is guaranteed. This function uses a static buffer that's overwritten by each call.



The `getgrent()`, `getgrgid()` (p. 853), and `getgrnam()` (p. 857) function share the same static buffer.

Returns:

The next entry from the group database. When you first call `getgrent()`, the group database is opened. It remains open until either `getgrent()` returns `NULL` to signify end-of-file, or you call `endgrent()` (p. 496).

Errors:

The `getgrent()` function uses the following functions, and as a result, `errno` (p. 539) can be set to an error for any of these calls:

- `fclose()` (p. 618)
- `fgets()` (p. 652)
- `fopen()` (p. 681)
- `fseek()` (p. 772)
- `rewind()` (p. 2646)

Examples:

```
/*
```

```
* This program loops, reading a group name from
* standard input and checking to see if it is a valid
* group. If it isn't valid, the entire contents of the
* group database are printed.
*/
#include <stdio.h>
#include <stdlib.h>
#include <grp.h>
#include <limits.h>

int main( void )
{
    struct group* gr;
    char    buf[80];

    setgrent();
    while( gets(buf) != NULL) {
        if( (gr=getgrnam(buf)) != (struct group *)0) {
            printf("Valid group is: %s\n",gr->gr_name);
        } else {
            setgrent();
            while( (gr=getgrent()) != (struct group *)0 )
                printf("%s\n",gr->gr_name);
        }
    }
    endgrent();
    return( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getgrgid()

Get information about the group with a given ID

Synopsis:

```
#include <sys/types.h>
#include <grp.h>

struct group* getgrgid( gid_t gid );
```

Arguments:

gid

The ID of the group you want to get information about.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getgrgid()* function lets a process gain more knowledge about group *gid*. This function uses a static buffer that's overwritten by each call.



The [getgrent\(\)](#) (p. 851), *getgrgid()*, and [getgrnam\(\)](#) (p. 857) functions share the same static buffer.

Returns:

A pointer to an object of type *struct group* containing an entry from the group database with a matching *gid*. On error or failure to find an entry with a matching *gid*, a NULL pointer is returned.

Examples:

```
/*
 * Print a list of all users in your group
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <grp.h>

int main( void )
```

```
{
    struct group* g;
    char** p;

    if( ( g = getgrgid( getgid() ) ) == NULL ) {
        fprintf( stderr, "getgrgid: NULL pointer\n" );
        return( EXIT_FAILURE );
    }
    printf( "group name:%s\n", g->gr_name );
    for( p = g->gr_mem; *p != NULL; p++ ) {
        printf( "\t%s\n", *p );
    }
    return( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getgrgid_r()

Get information about the group with a given ID

Synopsis:

```
#include <sys/types.h>
#include <grp.h>

int getgrgid_r ( gid_t gid,
                 struct group* grp,
                 char* buffer,
                 size_t bufsize,
                 struct group** result );
```

Arguments:

gid

The ID of the group you want to get information about.

grp

A pointer to a group structure where the function can store information about the group.

buffer

A buffer from which to allocate any memory required.

bufsize

The size of the buffer.

result

The address of a pointer that *getgrgid_r()* sets to the same pointer as *grp* on success, or to NULL if the function can't find the group.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, *getgrgid_r()* updates the group structure pointed by *grp* and stores a pointer to that structure at the location pointed

by *result*. The structure contains an entry from the group database with a matching *gid*.

This function allocates storage referenced by the group structure from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. You can determine the maximum size needed for this *buffer* by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETGR_R_SIZE_MAX`.

The *getgrgid_r()* stores a NULL pointer at the location pointed by *result* on error or if the requested entry isn't found.

Returns:

Zero for success, or an error number if an error occurred.

Errors:

ERANGE

Insufficient storage was supplied via *buffer* and *bufsize* to contain the resulting *group* structure.

The *getgrgid_r()* function uses the following functions, and as a result, [errno](#) (p. 539) can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)
- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getgrnam()

Get information about the group with a given name

Synopsis:

```
#include <sys/types.h>
#include <grp.h>

struct group* getgrnam( const char* name );
```

Arguments:

name

The name of the group you want to get information about.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgrnam()` function lets a process gain more knowledge about the group named *name*. This function uses a static buffer that's overwritten by each call.



The [getgrent\(\)](#) (p. 851), [getgrgid\(\)](#) (p. 853), and `getgrnam()` functions share the same static buffer.

Returns:

A pointer to an object of type `struct group` containing an entry from the group database with a matching name, or `NULL` on error or failure to find an entry with a matching name.

Examples:

```
/*
 * Print the name of all users in the group given in
 * argv[1]
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <grp.h>
```

```
int main( int argc, char** argv )
{
    struct group* g;
    char** p;

    if( ( g = getgrnam( argv[1] ) ) == NULL ) {
        fprintf( stderr, "getgrnam: %s failed\n",
            argv[1] );
        return( EXIT_FAILURE );
    }
    printf( "group name:%s\n", g->gr_name );
    for( p = g->gr_mem; *p != NULL; p++ ) {
        printf( "\t%s\n", *p );
    }
    return( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getgrnam_r()

Get information about the group with a given name

Synopsis:

```
#include <sys/types.h>
#include <grp.h>

int getgrnam_r( const char* name,
                struct group* grp,
                char* buffer,
                size_t bufsize,
                struct group** result );
```

Arguments:

name

The name of the group you want to get information about.

grp

A pointer to a group structure where the function can store information about the group.

buffer

A buffer from which to allocate any memory required.

bufsize

The size of the buffer.

result

The address of a pointer that *getgrgid_r()* sets to the same pointer as *grp* on success, or to NULL if the function can't find the group.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, the *getgrnam_r()* function updates the group structure pointed by *grp* and stores a pointer to that structure at the location

pointed by *result*. The structure contains an entry from the group database with a matching *name*.

This function allocates storage referenced by the group structure from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. You can determine the maximum size needed for this *buffer* by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETGR_R_SIZE_MAX`.

The *getgrnam_r()* stores a NULL pointer at the location pointed by *result* on error or if the requested entry isn't found.

Returns:

Zero for success, or an error number if an error occurred.

Errors:**ERANGE**

Insufficient storage was supplied via *buffer* and *bufsize* to contain the *group* structure.

The *getgrnam_r()* function uses the following functions, and as a result, [errno](#) (p. 539) can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)
- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getgrouplist()

Determine the group access list for a user

Synopsis:

```
#include <unistd.h>

int getgrouplist( const char *name,
                  gid_t basegid,
                  gid_t *groups,
                  int *ngroups );
```

Arguments:

name

The name of the user.

basegid

The *basegid* is automatically included in the list of groups. Typically this value is given as the group number from the password file.

groups

A pointer to an array where the function can store the group IDs.

ngroups

A pointer to the size of the *groups* array. The function sets the value pointed to by *ngroups* to be the actual number of groups found.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getgrouplist()* function reads the group file and determines the group access list for the user specified in *name*, automatically adding *basegid*.

Returns:

0

Success; the function fills in the group array and sets *ngroups to the number of groups found.

-1

The groups array is too small to hold all the user's groups. The function fills the group array with as many groups as fit.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <limits.h>

int main()
{
    int ngroups, i;
    gid_t *groups;

    ngroups = sysconf( _SC_NGROUPS_MAX );
    groups = (gid_t *) malloc ( ngroups * sizeof (gid_t) );
    if (groups == NULL) {
        perror ("malloc()");
        return EXIT_FAILURE;
    }

    if ( getgrouplist( getlogin(), getegid(), groups, &ngroups) == -1) {
        printf ("Groups array is too small: %d\n", ngroups);
    }

    printf ("%s belongs to these groups: %d", getlogin(), getegid());
    for (i=0; i < ngroups; i++) {
        printf (" ", %d", groups[i]);
    }
    printf ("\n");

    return EXIT_SUCCESS;
}
```

Files:

/etc/group

Group membership list.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

The *getgrouplist()* function uses the routines based on [getgrent\(\)](#) (p. 851). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *getgrouplist()*.

getgroups()

Get the supplementary group IDs of the calling process

Synopsis:

```
#include <unistd.h>

int getgroups( int gidsetsize,
               gid_t grouplist[] );
```

Arguments:

gidsetsize

The size of the *grouplist* array, or 0 if you want to get the number of supplementary group IDs.

grouplist

An array that the function can fill in with the process's supplementary group IDs.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getgroups()* function fills the array *grouplist* with the supplementary group IDs of the calling process. The maximum number of group IDs is given by `sysconf(_SC_NGROUPS_MAX)`. The values of array entries with indices greater than or equal to the returned value are undefined.

If *gidsetsize* is 0, the function returns the number of group IDs, without modifying the *grouplist* array.

Returns:

The number of supplementary groups IDs, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The *gidsetsize* argument isn't equal to zero, and is less than the number of supplementary group IDs.

Examples:

```
/*
 * Print the supplementary group IDs of
 * the calling process.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    int    gidsize;
    gid_t  *grouplist;
    int    i;

    gidsize = getgroups( 0, NULL );
    grouplist = malloc( gidsize * sizeof( gid_t ) );
    getgroups( gidsize, grouplist );
    for( i = 0; i < gidsize; i++ )
        printf( "%d\n", ( int ) grouplist[i] );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

gethostbyaddr()

Get a network host entry, given an Internet address

Synopsis:

```
#include <netdb.h>

struct hostent * gethostbyaddr( const void * addr,
                                socklen_t len,
                                int type );
```

Arguments:

addr

A pointer to the binary-format (i.e. not NULL-terminated) address in network byte order.

len

The length, in bytes, of *addr*.

type

The type of address. Currently, this must be AF_INET.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *gethostbyaddr()* function searches for information associated with a host, which has the address pointed to by *addr* within the address family specified by *type*, opening a connection to the database if necessary.



Both *gethostbyaddr()* and *gethostbyname()* are marked as obsolete in POSIX 1003.1. You should use [getaddrinfo\(\)](#) (p. 818) or [getnameinfo\(\)](#) (p. 894) instead.

This function returns a pointer to a structure of type [hostent](#) (p. 1019) that describes an Internet host. This structure contains either the information obtained from a name server, or broken-out fields from a line in `/etc/hosts`.

You can use [sethostent\(\)](#) (p. 2866) to request the use of a connected TCP socket for queries. If the *stayopen* flag is nonzero, all queries to the name server will use TCP

and the connection will be retained after each call to *gethostbyaddr()* or *gethostbyname()* (p. 872). If the *stayopen* flag is zero, queries use UDP datagrams.

Returns:

A pointer to a valid *hostent* structure, or NULL if an error occurs (*h_errno* (p. 1012) is set).

Errors:

See *herror()* (p. 1017).

Examples:

Use the *gethostbyaddr()* function to find a host:

```
struct sockaddr_in client;
struct hostent* host;

int sock, fd, len;

...

len = sizeof( client );
fd = accept( sock, (struct sockaddr*)&client, &len );
if( fd == -1 ) {
    perror( "accept" );
    exit( 1 );
}

host = gethostbyaddr( (const void*)&client.sin_addr,
                     sizeof(struct in_addr),
                     AF_INET );

printf( "Connection from %s: (%s)\n",
        host ? host->h_name : "<unknown>",
        inet_ntoa( client.sin_addr ) );

...
```

Files:

/etc/hosts

Host database file.

/etc/nsswitch.conf

Name-service switch configuration file.

Classification:

POSIX 1003.1 OBS (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it. Currently, only the Internet address format is understood.

gethostbyaddr_r()

Get a network host entry, in a thread-safe manner

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent * gethostbyaddr_r(
    const void * addr,
    socklen_t length,
    int type,
    struct hostent * result,
    char * buffer,
    int buflen,
    int * h_errnop );
```

Arguments:

addr

A pointer to the binary-format (i.e. not NULL-terminated) address in network byte order.

length

The length, in bytes, of *addr*.

type

The type of address. Currently, this must be AF_INET.

result

A pointer to a struct `hostent` where the function can store the host entry.

buffer

A pointer to a buffer that the function can use during the operation to store host database entries; *buffer* should be large enough to hold all of the data associated with the host entry. A 2K buffer is usually more than enough; a 256-byte buffer is safe in most cases.

buflen

The length of the area pointed to by *buffer*.

h_errnop

A pointer to a location where the function can store an [herrno](#) (p. 1012) value if an error occurs.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *gethostbyaddr_r()* function is a thread-safe version of [gethostbyaddr\(\)](#) (p. 866). This function gets the network host entry for the host specified by *addr*. The *addr* argument is the network address of the specified network family, *type*. The buffer for *addr* is at least *length* bytes.

If you need to convert a text-based address into the format necessary for use as *gethostbyaddr_r()*'s *addr*, see [inet_pton\(\)](#) (p. 1096).

Returns:

A pointer to *result*, or NULL if an error occurs.

Errors:

If an error occurs, the `int` pointed to by *h_errnop* is set to:

ERANGE

The supplied *buffer* isn't large enough to store the result.

HOST_NOT_FOUND

Authoritative answer: Unknown host.

NO_ADDRESS

No address associated with name; look for an MX record.

NO_DATA

Valid name, but no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn't a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

NO_RECOVERY

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

TRY_AGAIN

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn't receive a response from an authoritative server. A retry at some later time may succeed.

Files:**/etc/hosts**

Local host database file.

/etc/nsswitch.conf

Name-service switch configuration file.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

gethostbyname()*, *gethostbyname2()

Get a network host entry, given a name

Synopsis:

```
#include <netdb.h>

struct hostent * gethostbyname( const char * name );
struct hostent * gethostbyname2( const char * name,
                                int af );
```

Arguments:

name

The name of the Internet host whose entry you want to find.

af

(*gethostbyname2()* only) The address family; one of:

- AF_INET
- AF_INET6

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *gethostbyname()* routine gets the network host entry for a given name. It returns a pointer to a structure of type [hostent](#) (p. 1019) that describes an Internet host. This structure contains either the information obtained from a name server, or broken-out fields from a line in `/etc/hosts`.



Both *gethostbyaddr()* and *gethostbyname()* are marked as obsolete in POSIX 1003.1. You should use [getaddrinfo\(\)](#) (p. 818) or [getnameinfo\(\)](#) (p. 894) instead.

When using the name server, *gethostbyname()* searches for the named host in the current domain and in the domain's parents, unless the name ends in a dot.

- If the name server isn't responding, there's a timeout of 1.5 minutes per name server.



- If the name doesn't contain a dot, and the environment variable **HOSTALIASES** contains the name of an alias file, the alias file is first searched for an alias matching the input name. This file has the same form as `/etc/hosts`.

You can use [sethostent\(\)](#) (p. 2866) to request the use of a connected TCP socket for queries. If the *stayopen* flag is nonzero, all queries to the name server use TCP and the connection is retained after each call to *gethostbyname()* or [gethostbyaddr\(\)](#) (p. 866). If the *stayopen* flag is zero, queries use UDP datagrams.

The *gethostbyname2()* function is an evolution of the *gethostbyname()* function that lets you look up host names in address families other than `AF_INET`. If you specify an invalid address family, the function returns `NULL` and sets [h_errno](#) (p. 1012) to `NETDB_INTERNAL`.

Returns:

A pointer to a valid [hostent](#) (p. 1019) structure, or `NULL` if an error occurs ([h_errno](#) (p. 1012) is set).

Errors:

See [herror\(\)](#) (p. 1017).

Files:**`/etc/hosts`**

Host database file.

`/etc/nsswitch.conf`

Name-service switch configuration file.

For information about these files, see the *Utilities Reference*.

Environment variables:**`HOSTALIASES`**

Name of the alias file that *gethostbyname()* is to search first when the hostname doesn't contain a dot.

Classification:

gethostbyname() is [POSIX 1003.1 OBS](#) (p. 46); *gethostbyname2()* is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

gethostbyname_r()

Get a network host entry by name

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent *gethostbyname_r(
    const char * name,
    struct hostent * result,
    char * buffer,
    int buflen,
    int * h_errnop );
```

Arguments:

name

The name of the Internet host whose entry you want to find.

result

A pointer to a `struct hostent` where the function can store the host entry.

buffer

A pointer to a buffer that the function can use during the operation to store host database entries; *buffer* should be large enough to hold all of the data associated with the host entry. A 2K buffer is usually more than enough; a 256-byte buffer is safe in most cases.

buflen

The length of the area pointed to by *buffer*.

h_errnop

A pointer to a location where the function can store an [herrno](#) (p. 1012) value if an error occurs.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `gethostbyname_r()` function is a thread-safe version of [gethostbyname\(\)](#) (p. 872). This function gets the network host entry for the host specified by *name*, and stores the entry in the `struct hostent` pointed to by *result*.

Returns:

A pointer to *result*, or `NULL` if an error occurs.

Errors:

If an error occurs, the `int` pointed to by *h_errnop* is set to:

ERANGE

The supplied *buffer* isn't large enough to store the result.

HOST_NOT_FOUND

Authoritative answer: Unknown host.

NO_ADDRESS

No address associated with name; look for an MX record.

NO_DATA

Valid name, but no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn't a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

NO_RECOVERY

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

TRY_AGAIN

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn't receive a response from an authoritative server. A retry at some later time may succeed.

Files:

/etc/hosts

Local host database file.

/etc/nsswitch.conf

Name-service switch configuration file.

For information about these files, see the *Utilities Reference*.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

gethostent()

Read the next line of the host database file

Synopsis:

```
#include <netdb.h>

struct hostent * gethostent( void );
```

Library:

libsocket
Use the -l socket option to qcc to link against this library.

Description:

The *gethostent()* routine reads the next line in the host database file.

Returns:

A pointer to a valid `hostent` structure, or `NULL` if an error occurs.

Files:

- /etc/hosts**
Host database file.
- /etc/resolv.conf**
Resolver configuration file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, this function understands only the Internet address format.

gethostent_r()

Read the next line of the host database file

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent * gethostent_r( FILE ** hostf,
                               struct hostent * result,
                               char * buffer,
                               int buflen,
                               int * h_errnop );
```

Arguments:

hostf

NULL, or the address of the FILE * pointer associated with the host database file.

result

A pointer to a struct hostent where the function can store the host entry.

buffer

A pointer to a buffer that the function can use during the operation to store host database entries; *buffer* should be large enough to hold all of the data associated with the host entry. A 2K buffer is usually more than enough; a 256-byte buffer is safe in most cases.

buflen

The length of the area pointed to by *buffer*.

h_errnop

A pointer to a location where the function can store an [herrno](#) (p. 1012) value if an error occurs.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *gethostent_r()* function is a thread-safe version of the *gethostent()* (p. 878) function. This function gets the local host's entry. If the pointer pointed to by *hostf* is NULL, *gethostent_r()* opens */etc/hosts* and returns its file pointer in *hostf* for later use. It's the calling process's responsibility to close the host file with *fclose()* (p. 618).



The first time that you call *gethostent_r()*, pass NULL in the pointer pointed to by *hostf*.

Returns:

A pointer to *result*, or NULL if an error occurs.

Errors:

If an error occurs, the *int* pointed to by *h_errnop* is set to:

ERANGE

The supplied *buffer* isn't large enough to store the result.

HOST_NOT_FOUND

Authoritative answer: Unknown host.

NO_ADDRESS

No address associated with name, look for an MX record.

NO_DATA

Valid name, no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn't a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

NO_RECOVERY

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

TRY_AGAIN

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn't receive a response from an authoritative server. A retry at some later time may succeed.

Files:**/etc/hosts**

Local host database file.

Classification:*Unix* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

gethostname()

Get the name of the current host

Synopsis:

```
#include <unistd.h>

int gethostname( char * name,
                 size_t namelen );
```

Arguments:

name

A buffer where the function can store the host name.

namelen

The size of the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *gethostname()* function stores in *name* the standard hostname for the current processor, as previously set by [sethostname\(\)](#) (p. 2868). The parameter *namelen* specifies the size of the *name* array. The returned name is NULL-terminated unless insufficient space is provided.



This function gets the value of the `_CS_HOSTNAME` configuration string, not that of the ***HOSTNAME*** environment variable.

Returns:

0

Success.

-1

An error occurred (*errno* isn't set).

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Hostnames are limited to MAXHOSTNAMELEN characters (defined in <sys/param.h>).

getifaddrs()

Get a network interface address

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>

int getifaddrs( struct ifaddrs ** ifap );
```

Arguments:

ifap

The address of a location where the function can store a pointer to a linked list of `ifaddrs` structures that contain the data related to the network interfaces on the local machine.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *getifaddrs()* function stores a reference to a linked list of the network interfaces on the local machine in the memory referenced by *ifap*.

The data returned by *getifaddrs()* is dynamically allocated; you should free it by calling [freeifaddrs\(\)](#) (p. 720) when you no longer need it.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

The *getifaddrs()* function may fail and set [errno](#) (p. 539) for any of the errors specified by [ioctl\(\)](#) (p. 1198), [malloc\(\)](#) (p. 1569), [socket\(\)](#) (p. 3075), and [sysctl\(\)](#) (p. 3311).

It can also set *errno* to ENOMEM if the system is out of memory, or the interface list was growing while *getifaddrs()* was executing. Calling the function again when the interface list is stable can return 0 (success).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

GETIOVBASE()

Get the base member of an `iov_t` structure

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

#define GETIOVBASE( _iov ) ...
```

Arguments:

_iov

A pointer to the `iov_t` structure you want to get the base member from.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This macro evaluates to the `iov_base` member of the given `iov_t` structure.

Returns:

The `iov_base` member of the `iov_t` structure, which is of type `void *`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

GETIOVLEN()

Get the length member of an `iov_t` structure

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

#define GETIOVLEN( _iov ) ...
```

Arguments:

_iov

A pointer to the `iov_t` structure you want to get the length member from.

Library:

`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This macro evaluates to the *iov_len* member of the given `iov_t` structure.

Returns:

The *iov_len* member of the `iov_t` structure, which is of type `size_t`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

getitimer()

Get the value of an interval timer

Synopsis:

```
#include <sys/time.h>

int getitimer ( int which,
                struct itimerval *value );
```

Arguments:

which

The interval time whose value you want to get. Currently, this must be ITIMER_REAL.

value

A pointer to a `itimerval` structure where the function can store the value of the interval timer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The system provides each process with several interval timers, defined in `<sys/time.h>`. The *getitimer()* function stores the current value of the timer specified by *which* into the structure pointed to by *value*.

A timer value is defined by the `itimerval` structure, which includes the following members:

```
struct timeval  it_interval;    /* timer interval */
struct timeval  it_value;       /* current value */
```

Each `struct timeval` contains the following members:

- `time_t tv_sec` — the number of seconds since the start of the Unix Epoch.
- `suseconds_t tv_usec` — the number of microseconds.

The *it_value* member indicates the time to the next timer expiration. The *it_interval* member specifies a value to be used in reloading *it_value* when the timer expires.

Setting *it_value* to 0 disables a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 disables a timer after its next expiration (assuming *it_value* is nonzero).

Time values smaller than the resolution of the system clock are rounded up to the resolution of the system clock.

The interval timers include:

ITIMER_REAL

Decrements in real time. A SIGALRM signal is delivered when this timer expires.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the *which* argument is unrecognized.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getlogin()

Get the user name associated with the calling process

Synopsis:

```
#include <unistd.h>

char* getlogin( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getlogin()* function returns a pointer to a string containing the login name of the user associated with the calling process.

Returns:

A pointer to a string containing the user's login name, or `NULL` if the user's login name can't be found.

The return value from *getlogin()* may point to static data and, therefore, may be overwritten by each call.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getlogin_r()

Get the user name associated with the calling process

Synopsis:

```
#include <unistd.h>

int getlogin_r( char* name,
               size_t namesize );
```

Arguments:

name

A buffer where the function can store the user name.

namesize

The size of the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, the `getlogin_r()` function puts the login name of the user associated with the calling process in the character array pointed to by *name*. The array is *namesize* characters long and should have space for the name and the terminating NULL character. The maximum size of the login name is `_POSIX_LOGIN_NAME_MAX`.

If `getlogin_r()` is successful, *name* points to the name the user used at login, even if there are several login names with the same user ID.

Returns:

EOK

Success.

ERANGE

Insufficient storage was supplied via the *name* and *namesize* arguments to contain the user's name.

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getnameinfo()

Perform address-to-nodename translation in a protocol-independent manner

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa,
                socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen,
                int flags);
```

Arguments:

sa

Points to either a `sockaddr_in` structure (for IPv4) or a `sockaddr_in6` structure (for IPv6) that holds the IP address and port number.

salen

Length of the `sockaddr_in` or `sockaddr_in6` structure.

host

Buffer pointer for the host.

hostlen

Size of the host buffer.

serv

Buffer pointer for the server.

servlen

Length of the server buffer.

flags

Change the default action of *getnameinfo()*. By default, the fully qualified domain name (FQDN) for the host is looked up in the DNS and returned.

These flags are defined in `<netdb.h>`:

NI_NOFQDN

Only the nodename portion of the FQDN is returned for local hosts.

NI_NUMERICHOST

If set, or if the host's name can't be located in the DNS, the numeric form of the host's address is returned instead of its name (e.g. by calling *inet_ntop()* instead of *getnodebyaddr()*).

NI_NAMEREQD

If set, an error is returned when the host's name can't be located in the DNS.

NI_NUMERICSERV

If set, the numeric form of the service address (instead of its name) is returned e.g. its port number. You may require two **NI_NUMERICxxx** flags to support the **-n** flag that many commands provide.

NI_DGRAM

Specify that the service is a datagram service. Call *getservbyport()* with a second argument of *udp* instead of its default of *tcp*. This is required for the few ports (512-514) that have different services for UDP and TCP.

Library:

libsocket

Use the **-l socket** option to **qcc** to link against this library.

Description:

The *getnameinfo()* function defines and performs protocol-independent address-to-nodename translation. You can think of it as implementing the reverse-functionality of *getaddrinfo()* or similar functionality of *gethostbyaddr()* or *getservbyport()*.

This function looks up an IP address and port number provided by the caller in the DNS and system-specific database. For both IP address and port number, the *getnameinfo()* function returns text strings in respective buffers provided by the caller. The function indicates successful completion by a zero return value; a non-zero return value indicates failure.

The *getnameinfo()* function returns the nodename associated with the IP address in the buffer pointed to by the *host* argument. The *hostlen* argument gives the length of this buffer.

The *getnameinfo()* function returns the service name associated with the port number in the buffer pointed to by the *serv* argument. The *servlen* argument gives the length of this buffer.

Specify zero for *hostlen* or *servlen* when the caller chooses not to return either string. Otherwise, the caller must provide buffers large enough to hold the nodename and the service name, including the terminating null characters.

Most systems don't provide constants that specify the maximum size of either a FQDN or a service name. In order to aid your application in allocating buffers, the following constants are defined in `<netdb.h>`:

```
#define NI_MAXHOST 1025
#define NI_MAXSERV 32
```

You may find the first value as the constant `MAXDNAME` in recent versions of BIND's `<arpa/nameser.h>`; older versions of BIND define this constant to be 256. The second value is a guess based on the services listed in the current Assigned Numbers RFC. BIND (Berkeley Internet Name Domain) is a suite of functionalities that implements Domain Name System (DNS) protocols.

Extension

The implementation allows experimental numeric IPv6 address notation with scope identifier. An IPv6 link-local address appears as string like `fe80::1%ne0`, when the `NI_WITHSCOPEID` bit is enabled in the *flags* argument. See *getaddrinfo()* for the notation.

Returns:

0

Success.

Non-zero value

An error occurred (see below).

Errors:

EAI_AGAIN

The name couldn't be resolved at this time. Future attempts may succeed.

EAI_BADFLAGS

The flags had invalid values.

EAI_FAIL

A nonrecoverable error occurred.

EAI_FAMILY

The address family wasn't recognized or the address length was invalid for the specified family.

EAI_MEMORY

There was a memory allocation failure.

EAI_NONAME

The name doesn't resolve for the supplied parameters. `NI_NAMEREQD` is set and the host's name can't be located, or both *node* name and *serv* name were null.

EAI_SYSTEM

A system error occurred. The error code can be found in *errno*.

Examples:

The following code gets the numeric hostname and the service name for a given socket address. There is no hardcoded reference to a particular address family.

```
struct sockaddr *sa;    /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), sbuf,
    sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
    errx(1, "could not get numeric hostname");
    /*NOTREACHED*/
}
printf("host=%s, serv=%s\n", hbuf, sbuf);
```

The following version checks if the socket address has reverse address mapping.

```
struct sockaddr *sa;    /* input */
char hbuf[NI_MAXHOST];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), NULL, 0,
    NI_NAMEREQD)) {
    errx(1, "could not resolve hostname"); /*NOTREACHED*/
}
printf("host=%s\n", hbuf);
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getnetbyaddr()

Get a network entry, given an address (Unix)

Synopsis:

```
#include <netdb.h>

struct netent * getnetbyaddr( uint32_t net,
                             int type );
```

Arguments:

net

The net address whose network entry you want to find.

type

The address type. This must currently be AF_INET.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getnetbyaddr()* function gets an entry for the given address, *net*, from the network database, */etc/networks*.

This function returns a pointer to a structure of type [netent](#) (p. 1876), which contains the broken-out fields of a line in the network database.

The [setnetent\(\)](#) (p. 2884) function opens and rewinds the file. If you pass a nonzero *stayopen* argument to *setnetent()*, the network database isn't closed after each call to [getnetbyname\(\)](#) (p. 900), or *getnetbyaddr()*.

The *getnetbyname()* and *getnetbyaddr()* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

Returns:

A pointer to a valid *netent* structure, or NULL if an error occurs.

Files:

/etc/networks

Network name database file.

/etc/nsswitch.conf

Name-service switch configuration file.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Only Internet network numbers are currently understood.

getnetbyname()

Get a network entry, given a name

Synopsis:

```
#include <netdb.h>

struct netent * getnetbyname( const char * name );
```

Arguments:

name

The name of the network whose entry you want to find.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getnetbyname()* function gets the network entry for the given name. This function returns a pointer to a structure of type [netent](#) (p. 1876), which contains the broken-out fields of a line in the network database, */etc/networks*.

The [setnetent\(\)](#) (p. 2884) function opens and rewinds the file. If you pass a nonzero *stayopen* argument to *setnetent()*, the network database isn't closed after each call to *getnetbyname()* or [getnetbyaddr\(\)](#) (p. 898).

The *getnetbyaddr()* and *getnetbyname()* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

Returns:

A pointer to a valid netent structure, or NULL if an error occurs.

Files:

/etc/networks

Network name database file.

/etc/nsswitch.conf

Name-service switch configuration file.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getnetent()

Read the next line of the network name database file

Synopsis:

```
#include <netdb.h>

struct netent * getnetent( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getnetent()* function reads the next line of the network name database file, opening the file if necessary. It returns a pointer to a structure of type [netent](#) (p. 1876), which contains the broken-out fields of a line in the network database, */etc/networks*.

Returns:

A pointer to a valid netent structure, or NULL if an error occurs.

Files:

/etc/networks

Network name database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getopt()

Parse options from a command line

Synopsis:

```
#include <unistd.h>

int getopt( int argc,
            char * const argv[],
            const char * optstring );

extern char * optarg;
extern int optind, opterr, optopt;
```

Arguments:

argc

The argument count that was passed to *main()*.

argv

The argument array that was passed to *main()*.

optstring

A string of recognized option letters; if a letter is followed by a colon, the option takes an argument. Valid option characters for *optstring* consist of a single alphanumeric character (i.e. a letter or digit).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getopt()* function is a command-line parser that can be used by applications that follow the [Utility Syntax Guidelines](#) (p. 904) described below.

The *optind* global variable is the index of the next element of the *argv[]* vector to be processed. The system initializes *optind* to 1 when the program is loaded, and *getopt()* updates it when it finishes with each element of *argv[]*. Reset *optind* to 1 if you want to use *getopt()* to process additional argument sets.

The *getopt()* function returns the next option character from *argv* that matches a letter in *optstring*, if there's one that matches. If the option takes an argument, *getopt()* sets the global variable *optarg* to point to the option argument as follows:

1. If the option is the last letter in the string pointed to by an element of *argv*, then *optarg* contains the next element of *argv*, and *optind* is incremented by 2.
2. Otherwise, *optarg* points to the string following the option letter in that element of *argv*, and *optind* is incremented by 1.

The *getopt()* function returns -1 is returned and doesn't change *optind* if:

- *argv[optind]* is NULL
- **argv[optind]* isn't the character ' - '
- *argv[optind]* points to the string " - ".

This function returns -1 after incrementing *optind*, if:

- *argv[optind]* points to the string " - - ".

If *getopt()* encounters an option character that isn't contained in *optstring*, it returns the ? character. If it detects a missing option argument, *getopt()* returns:

- a colon (:) if the first character of *optstring* is a colon
- or:
- a question mark (?) otherwise

and sets *optopt* to the option character that caused the error.

The *getopt()* always prints a diagnostic message to *stderr* (p. 3175) unless *opterr* is set to 0, or the first character of *optstring* is a : character.

Utility Syntax Guidelines

The *getopt()* function may be used by applications that follow these guidelines:

- When describing the syntax of a utility, the options are listed in alphabetical order. There's no implied relationship between the options based upon the order in which they appear, unless otherwise stated in the Options section, or:
 - the options are documented as mutually-exclusive and such an option is documented to override any incompatible options preceding it
 - when an option has option arguments repeated, the option and option argument combinations are interpreted in the order specified on the command line.

If an option that doesn't have option arguments is repeated, the results depend on the application.

- Names of parameters that require substitution by actual values may be shown with embedded underscores or as *<parameter name>*. Angle brackets are used for the symbolic grouping of a phrase representing a single parameter and portable applications shouldn't include them in data submitted to the utility.

- Options may be documented individually, or grouped (if they don't take option arguments):

```
utility_name [-a] [-b] [-c option_argument]
              [-d|-e] [-foption_argument] [operand...]
```

Or:

```
utility_name [-ab] [-c option_argument]
              [-d|-e] [-foption_argument] [operand...]
```

Utilities with very complex arguments may be shown as:

```
utility_name [options] [operand]
```

- Unless specified, whenever an operand or option argument is, or contains, a numeric value:
 - the number is interpreted as a decimal integer
 - numerals in the range 0 to 2,147,483,647 are syntactically recognized as numeric values
 - when the utility description states that it accepts negative numbers as operands or option arguments, numerals in the range -2,147,483,647 to 2,147,483,647 are syntactically recognized as numeric values
 - ranges greater than those listed here are allowed.

All numbers within the allowable range aren't necessarily semantically correct. A standard utility that accepts an option argument or operand that's to be interpreted as a number, and for which a range of values smaller than that shown above is permitted, describes that smaller range along with the description of the option argument or operand. If an error is generated, the utility's diagnostic message indicates that the value is out of the supported range, not that it's syntactically incorrect.

- Arguments or option arguments enclosed in the “[” and “]” notation are optional and can be omitted. Portable applications shouldn't include the “[” and “]” symbols in data submitted to the utility.
- Ellipses (...) are used to denote that one or more occurrences of an option or operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands may be specified. The forms:

```
utility_name -f option_argument...[operand...]
utility_name [-g option_argument]...[operand...]
```

indicate that multiple occurrences of the option and its option argument preceding the ellipses are valid, with semantics as indicated in the Options section of the

utility. In the first example, each option argument requires a preceding -f and at least one *-foption_argument* must be given.

- When the synopsis is too long to be printed on a single line in the documentation, the indented lines following the initial line are continuation lines. An actual use of the command appears on a single logical line.

See also Guidelines 3,4,5,6,7,9 and 10 in the *Base Definitions* volume of the IEEE Std.1003.1-2001 Section 12.2, *Utility Syntax Guidelines*.

Returns:

One of the following:

- the next option character specified on the command line
- a colon if a missing argument is detected and the first character of *optstring* is a colon
- a question mark if a missing argument is detected and the first character of *optstring* isn't a colon
- a question mark if an option character is encountered that's not in *optstring*
- -1 when all command-line options have been parsed

Examples:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int c, errflag = 0;

    while( ( c = getopt( argc, argv, "abt:" ) )
        != -1 ) {
        switch( c ) {
            case 'a': printf( "apples\n" );
                      break;
            case 'b': printf( "bananas\n" );
                      break;
            case 't': printf( "tree = %s\n", optarg );
                      break;
            case '?': ++errflag;
                      break;
        }
    }
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

Caveats:

A cancellation point may occur when a thread is executing *getopt()* if the *opterr* argument is nonzero.

getpagesize()

Get the current page size (Legacy Unix)

Synopsis:

```
#include <unistd.h>

int getpagesize(void);
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *getpagesize()* function gets the current page size. It's equivalent to:

```
sysconf(_SC_PAGESIZE)
```



The value that *getpagesize()* returns might not be the minimum value that *malloc()* (p. 1569) can allocate, and you shouldn't assume you can actually allocate an object of this size.

Because *getpagesize()* returns an `int`, it might not represent large values correctly. You should use *sysconf()* (p. 3308) instead.

Returns:

The current page size.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getpagesizes(), getpagesizes64()

Get the available page sizes

Synopsis:

```
#include <sys/mman.h>

int getpagesizes( size_t pagesize[],
                  int nelem );

int getpagesizes64( uint64_t pagesize[],
                    int nelem );
```

Arguments:***pagesize***

NULL, or an array where the function can store the available page sizes.

nelem

The number of elements in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getpagesizes()* and *getpagesizes64()* functions get the available page sizes and store up to *nelem* of them in the *pagesize* array. Software that wants to optimize its usage of large pages can use these functions to determine what sizes are available.

If you use *getpagesizes64()*, some page sizes will have the `_SYS_GETPAGESIZES_HIGHUSAGE` bit (defined in `<sys/sysmsg.h>`) set to indicate that they're available only when mapping objects marked with the `SHMCTL_HIGHUSAGE` flag. These page sizes aren't included in the list provided by *getpagesizes()*.

If you pass NULL for *pagesize* and 0 for *nelem*, the functions return the number of available page sizes.

Returns:

The number of entries filled in the array, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:**EINVAL**

The value of the *nelem* argument is negative, or it's positive, but *pagesize* is NULL.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main ( void )
{
    size_t *pagesizes;
    int i, num_found;

    num_found = getpagesizes( NULL, 0 );
    if (num_found <= 0)
    {
        printf ("num_found is %d\n", num_found);
        return EXIT_FAILURE;
    }

    pagesizes = (size_t *) malloc ( num_found * sizeof (size_t));
    if (pagesizes == NULL)
    {
        perror ("malloc()");
        return EXIT_FAILURE;
    }

    num_found = getpagesizes( pagesizes, NUM_SIZES );
    printf ("Found %d sizes:\n", num_found);
    for (i=0; i < num_found; i++)
    {
        printf ("    %d\n", pagesizes[i]);
    }
    return EXIT_SUCCESS;
}
```

Classification:

getpagesizes() is [Unix](#) (p. 46); *getpagesizes64()* is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getpass()

Prompt for and read a password

Synopsis:

```
#include <unistd.h>

char *getpass( const char *prompt );
```

Arguments:

prompt

The string you want to display to prompt for the password.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getpass()` function can be used to get a password. It opens the current terminal, displays the given *prompt*, suppresses echoing, reads up to 32 characters into a static buffer, and restores echoing. This function adds a null character to the end of the string, but ignores additional characters and the newline character.

Returns:

A pointer to the static buffer.

Classification:

[Legacy Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

Caveats:

This function leaves its result in an internal static buffer and returns a pointer to it. Subsequent calls to *getpass()* modify the same buffer. The calling process should zero the password as soon as possible to avoid leaving the clear-text password visible in the process's address space.

getpeereid()

Get the effective credentials of a UNIX-domain peer

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

int getpeereid( int s,
                uid_t *euid,
                gid_t *egid );
```

Arguments:

s

A UNIX-domain socket (see the [UNIX](#) (p. 3577) protocol) of type `SOCK_STREAM` on which either you've called [connect\(\)](#) (p. 348), or one returned from [accept\(\)](#) (p. 58) after you've called [bind\(\)](#) (p. 208) and [listen\(\)](#) (p. 1523).

euid

NULL, or a pointer to a location where the function can store the effective user ID.

egid

NULL, or a pointer to a location where the function can store the effective group ID.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getpeereid()* function gets the effective user and group IDs of the peer connected to a UNIX-domain socket. If *euid* and *egid* are non-NULL, the function stores the IDs in the locations they point to.

If you got the socket by calling *connect()*, the credentials are those of your peer when you called *bind()*. If the socket was one returned from *accept()*, the credentials are those of your peer when you called *connect()*. This mechanism is reliable; there is no way for either side to influence the credentials returned to its peer except by calling

the appropriate system call (i.e., either *connect()* or *bind()*) under different effective credentials.

UNIX-domain servers and clients commonly use this function to verify each other's credentials.

Returns:

0

Success.

-1

An error occurred; *errno* (p. 539) is set.

Errors:

EBADF

The argument *s* isn't a valid descriptor.

ENOTSOCK

The argument *s* is a file, not a socket.

ENOTCONN

The argument *s* doesn't refer to a socket on which you've called *connect()*, or isn't one returned by *listen()*.

EINVAL

The argument *s* doesn't refer to a socket of type SOCK_STREAM, or the system returned invalid data.

Classification:

NetBSD (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getpeername()

Get the name of the peer connected to a socket

Synopsis:

```
#include <sys/socket.h>

int getpeername( int s,
                 struct sockaddr * name,
                 socklen_t * namelen );
```

Arguments:

s

The socket whose connected peer you want to get.

name

A buffer where the function can store the name of the peer.

namelen

A pointer to a `socklen_t` object that initially specifies the size of the buffer. This function stores the actual size of the name, in bytes, in this object.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *getpeername()* function returns the name of the peer connected to socket *s*. The name is truncated if the buffer provided is too small.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid descriptor *s*.

EFAULT

The *name* parameter points to memory *not* in a valid part of the process address space.

ENOBUFS

Insufficient resources were available in the system to perform the operation.

ENOTCONN

The socket isn't connected.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

getpgid()

Get a process group ID

Synopsis:

```
#include <unistd.h>

pid_t getpgid( pid_t pid );
```

Arguments:

pid

The ID of the process whose process group ID you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getpgid()* returns the group ID for the process specified by *pid*. If *pid* is 0, *getpgid()* returns the calling process's group ID.



In order to get the group ID of a process outside the calling process's session, your process must have the `PROCMGR_AID_GETID` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The following definitions are worth mentioning:

Process

An executing instance of a program, identified by a nonnegative integer called a process ID.

Process group

A collection of one or more processes, with a unique process group ID. A process group ID is a positive integer.

Returns:

A process group ID for success, or `(pid_t)-1` if an error occurs.

Errors:

If an error occurs, [errno](#) (p. 539) is set to:

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process specified by *pid* doesn't exist.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getpgrp()*Get the process group***Synopsis:**

```
#include <sys/types.h>
#include <process.h>

pid_t getpgrp( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *getpgrp()* function gets the ID of the process group to which the calling process belongs.

Returns:

The calling process's process group ID.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <sys/types.h>

int main( void )
{
    printf( "I am in process group %d\n", (int) getpgrp() );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getpid()

Get the process ID

Synopsis:

```
#include <process.h>

pid_t getpid( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getpid()* function gets the process ID for the calling process.

Returns:

The process ID of the calling process.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

int main ( void )
{
    printf( "I'm process %d\n", getpid() );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getppid()*Get the parent process ID***Synopsis:**

```
#include <sys/types.h>
#include <process.h>

pid_t getppid( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *getppid()* function gets the process ID of the parent of the calling process.

Returns:

The calling process's parent's process ID.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <process.h>

int main( void )
{
    printf( "My parent is %d\n", getppid() );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getprio()

Get the priority of a given process

Synopsis:

```
#include <sched.h>

int getprio( pid_t pid );
```

Arguments:

pid

The process ID of the process whose priority you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getprio()* function returns the current priority of thread 1 in process *pid*. If *pid* is zero, the priority of the calling thread is returned.

Returns:

The priority, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

ESRCH

The process *pid* doesn't exist.

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

The *getprio()* and *setprio()* functions are included in the libraries for porting QNX 4 applications. For new programs, use [pthread_getschedparam\(\)](#) (p. 2299).

getprotobyname()

Get a protocol entry, given a name

Synopsis:

```
#include <netdb.h>

struct protoent * getprotobyname( const char * name );
```

Arguments:

name

The name of the protocol whose entry you want to get.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *getprotobyname()* function gets the entry for the given name from the protocol database, `/etc/protocols`. This function returns a pointer to a structure of type [protoent](#) (p. 2184), which contains the broken-out fields of a line in the network protocol database.

The [setprotoent\(\)](#) (p. 2892) function opens and rewinds the file. If you pass a nonzero *stayopen* argument to *setprotoent()*, the protocol database isn't closed after each call to *getprotobyname()* or [getprotobynumber\(\)](#) (p. 926).

The *getprotobyname()* and *getprotobynumber()* functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

Returns:

A pointer to a valid `protoent` structure, or NULL if an error occurs.

Files:

`/etc/protocols`

Protocol name database file.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

getprotobynumber()

Get a protocol entry, given a number

Synopsis:

```
#include <netdb.h>

struct protoent * getprotobynumber( int proto );
```

Arguments:

proto

The protocol number whose entry you want to get.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getprotobynumber()* function gets the protocol entry for the given number. It returns a pointer to structure of type [protoent](#) (p. 2184), which contains the broken-out fields of a line in the network protocol database, */etc/protocols*.

The [setprotoent\(\)](#) (p. 2892) function opens and rewinds the file. If you pass a nonzero *stayopen* argument to *setprotoent()*, the protocol database isn't closed after each call to [getprotobyname\(\)](#) (p. 924) or *getprotobynumber()*.

The *getprotobyname()* and *getprotobynumber()* functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

Returns:

A pointer to a valid *protoent* structure, or NULL if an error occurs.

Files:

/etc/protocols

Protocol name database file.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

getprotoent()

Read the next line of the protocol name database file

Synopsis:

```
#include <netdb.h>

struct protoent * getprotoent( void );
```

Library:

libsocket
Use the -l socket option to qcc to link against this library.

Description:

The *getprotoent()* function reads the next line of the protocol name database file, opening the file if necessary. It returns a pointer to a structure of type [protoent](#) (p. 2184), which contains the broken-out fields of a line in the network protocol database, */etc/protocols*.

Returns:

A pointer to a valid *protoent* structure, or NULL if an error occurs.

Files:

/etc/protocols
Protocol name database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

getpwent()

Get the next entry from the password database

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwent( void );
```

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *getpwent()* function returns the next entry from the password database. This function uses a static buffer that's overwritten by each call.



The *getpwent()*, [getpwnam\(\)](#) (p. 934), and [getpwuid\(\)](#) (p. 938), functions share the same static buffer. For a reentrant version of this function, see [getpwent_r\(\)](#) (p. 932).

Returns:

A pointer to an object of type `struct passwd` containing the next entry from the password database. When *getpwent()* is first called, the password database is opened, and remains open until either a `NULL` is returned to signify end-of-file, or *endpwent()* is called.

Errors:

The *getpwent()* function uses the following functions, and as a result, [errno](#) (p. 539) can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)
- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Examples:

```

/*
 * This program loops, reading a login name from standard
 * input and checking to see if it is a valid name. If it
 * is not valid, the entire contents of the name in the
 * password database are printed.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main( void )
{
    struct passwd* pw;
    char    buf[80];

    setpwent( );
    while( gets( buf ) != NULL ) {
        if( ( pw = getpwnam( buf ) ) != ( struct passwd * )0 ) {
            printf( "Valid login name is: %s\n", pw->pw_name );
        } else {
            setpwent( );
            while( ( pw=getpwent( ) ) != ( struct passwd * )0 )
                printf( "%s\n", pw->pw_name );
        }
    }
    endpwent();
    return( EXIT_SUCCESS );
}

```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getpwent_r()

Get the next entry from the password database

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

int getpwent_r( struct passwd *pwd,
                char *buffer,
                size_t bufsize,
                struct passwd **result);
```

Arguments:

pwd

A pointer to a `passwd` structure where the function can store the entry.

buffer

A block of memory that the function can use to allocate storage referenced by the `passwd` structure. You can determine the maximum size needed for this buffer by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETPW_R_SIZE_MAX`.

bufsize

The size of the block that *buffer* points to, in characters.

result

The address of a pointer to a `passwd` structure. If *getpwent_r()* finds the next entry, it stores a pointer to *pwd* in the location indicated by *result*; otherwise the function stores a NULL pointer there.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getpwent_r()* function returns the next entry from the password database. It's similar to [getpwent\(\)](#) (p. 930), but *getpwent_r()* is reentrant.

Returns:

0 on success or if the entry couldn't be found, or nonzero if an error occurred ([errno](#) (p. 539) is set).

Errors:

The *getpwent_r()* function uses the following functions, and as a result, [errno](#) (p. 539) can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)
- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

getpwnam()

Get information about the user with a given name

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwnam( const char* name );
```

Arguments:

name

The name of the user whose entry you want to find.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getpwnam()` function gets information about the user with the given *name*. It uses a static buffer that's overwritten by each call.



The [getpwent\(\)](#) (p. 930), `getpwnam()`, and [getpwuid\(\)](#) (p. 938) functions share the same static buffer.

The [getpwnam_r\(\)](#) (p. 936) function is a reentrant version of `getpwnam()`.

Returns:

A pointer to an object of type `struct passwd` containing an entry from the group database with a matching *name*. A `NULL` pointer is returned on error or failure to find a entry with a matching *name*.

Examples:

```
/*
 * Print information from the password entry
 * about the user name given as argv[1].
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
```

```

#include <pwd.h>

int main( int argc, char* *argv )
{
    struct passwd* pw;

    if( ( pw = getpwnam( argv[1] ) ) == NULL ) {
        fprintf( stderr, "getpwnam: unknown %s\n",
            argv[1] );
        return( EXIT_FAILURE );
    }
    printf( "login name  %s\n", pw->pw_name );
    printf( "user id     %d\n", pw->pw_uid );
    printf( "group id    %d\n", pw->pw_gid );
    printf( "home dir    %s\n", pw->pw_dir );
    printf( "login shell %s\n", pw->pw_shell );
    return( EXIT_SUCCESS );
}

```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getpwnam_r()

Get information about the user with a given name

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r( const char* name,
                struct passwd* pwd,
                char* buffer,
                size_t bufsize,
                struct passwd* result );
```

Arguments:

name

The name of the user whose entry you want to find.

pwd

A pointer to a `passwd` structure where the function can store the entry.

buffer

A block of memory that the function can use to allocate storage referenced by the `passwd` structure. You can determine the maximum size needed for this buffer by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETPW_R_SIZE_MAX`.

bufsize

The size of the block that *buffer* points to, in characters.

result

The address of a pointer to a `passwd` structure. If *getpwnam_r()* finds the entry, it stores a pointer to *pwd* in the location indicated by *result*; otherwise the function stores a NULL pointer there.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *getpwnam_r()* function is a reentrant version of *getpwnam()* (p. 934). It gets information about the user with the given *name*.

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, the *getpwnam_r()* function updates the `passwd` structure pointed to by *pwd* and stores a pointer to that structure at the location pointed by *result*. The structure contains an entry from the user database with the given *name*.

The function stores a `NULL` pointer at the location pointed by *result* on error or if it can't find the requested entry.

Returns:

Zero for success, or an error number.

Errors:**ERANGE**

Insufficient storage was supplied via *buffer* and *bufsize* to contain the resulting *passwd* structure.

The *getpwnam_r()* function uses the following functions, and as a result, *errno* (p. 539) can be set to an error for any of these calls:

- *fclose()* (p. 618)
- *fgets()* (p. 652)
- *fopen()* (p. 681)
- *fseek()* (p. 772)
- *rewind()* (p. 2646)

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getpwuid()

Get information about the user with a given ID

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwuid( uid_t uid );
```

Arguments:

uid

The userid whose entry you want to find.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getpwuid()* function gets information about user *uid*. This function uses a static buffer that's overwritten by each call.



The [getpwent\(\)](#) (p. 930), [getpwnam\(\)](#) (p. 934), and *getpwuid()* functions share the same static buffer.

Returns:

A pointer to an object of type `struct passwd` containing an entry from the group database with a matching *uid*, or `NULL` if an error occurred or the function couldn't find a matching entry.

Examples:

```
/*
 * Print password info on the current user.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main( void )
```

```

{
    struct passwd* pw;

    if( ( pw = getpwuid( getuid() ) ) == NULL ) {
        fprintf( stderr,
            "getpwuid: no password entry\n" );
        return( EXIT_FAILURE );
    }
    printf( "login name  %s\n", pw->pw_name );
    printf( "user id     %d\n", pw->pw_uid );
    printf( "group id    %d\n", pw->pw_gid );
    printf( "home dir    %s\n", pw->pw_dir );
    printf( "login shell %s\n", pw->pw_shell );
    return( EXIT_SUCCESS );
}

```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

getpwuid_r()

Get information about the user with a given ID

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r( uid_t uid,
                struct passwd* pwd,
                char* buffer,
                size_t bufsize,
                struct passwd** result );
```

Arguments:

uid

The userid whose entry you want to find.

pwd

A pointer to a `passwd` structure where the function can store the entry.

buffer

A block of memory that the function can use to allocate storage referenced by the `passwd` structure. You can determine the maximum size needed for this buffer by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETPW_R_SIZE_MAX`.

bufsize

The size of the block that *buffer* points to, in characters.

result

The address of a pointer to a `passwd` structure. If *getpwnam_r()* finds the entry, it stores a pointer to *pwd* in the location indicated by *result*; otherwise the function stores a NULL pointer there.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *getpwuid_r()* function is a reentrant version of *getpwuid()* (p. 938). It lets a process gain more knowledge about user with the given *uid*.

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, the *getpwuid_r()* function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure at the location pointed by *result*. The structure contains an entry from the user database with a matching *uid*.

The function stores a `NULL` pointer at the location pointed by *result* on error or if it can't find the requested entry.

Returns:

Zero for success, or an error number.

Errors:**ERANGE**

Insufficient storage was supplied via *buffer* and *bufsize* to contain the resulting *passwd* structure.

The *getpwuid_r()* function uses the following functions, and as a result, *errno* (p. 539) can be set to an error for any of these calls:

- *fclose()* (p. 618)
- *fgets()* (p. 652)
- *fopen()* (p. 681)
- *fseek()* (p. 772)
- *rewind()* (p. 2646)

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getrlimit(), getrlimit64()

Get the limit on a system resource

Synopsis:

```
#include <sys/resource.h>

int getrlimit( int resource,
               struct rlimit * rlp );

int getrlimit64( int resource,
                 struct rlimit64 * rlp );
```

Arguments:

resource

The resource whose limit you want to get; one of the following:

- RLIMIT_AS
- RLIMIT_CORE
- RLIMIT_CPU
- RLIMIT_DATA
- RLIMIT_FREEMEM
- RLIMIT_MEMLOCK
- RLIMIT_NOFILE
- RLIMIT_NPROC
- RLIMIT_NTHR
- RLIMIT_OFILE
- RLIMIT_RSS
- RLIMIT_STACK
- RLIMIT_VMEM

For descriptions and the actions taken when the current limit is exceeded, see [setrlimit\(\)](#) (p. 2899).

rlp

A pointer to a `rlimit` or `rlimit64` structure where the function can store the limit on the resource. The `rlimit` and `rlimit64` structures include at least the following members:

```
rlim_t rlim_cur; /* current (soft) limit */
rlim_t rlim_max; /* hard limit */
```

The `rlim_t` type is an arithmetic data type to which you can cast objects of type `int`, `size_t`, and `off_t` without loss of information.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getrlimit()` and `getrlimit64()` functions get the limits on the consumption of a variety of system resources by a process and each process it creates. The `getrlimit64()` function is a 64-bit version of `getrlimit()`.

Each call to `getrlimit()` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values:

- the current (soft) limit
- a maximum (hard) limit.

A process can change soft limits to any value that's less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that's greater than or equal to the soft limit. Only a process with an effective user ID of `root` can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit()` subject to the constraints described above. Limits may have an “infinite” value of `RLIM_INFINITY`.

Because limit information is stored in the per-process information, the shell builtin `ulimit` command (see the entry for `ksh` in the *Utilities Reference*) must directly execute this system call if it's to affect all future processes created by the shell.

The values of the current limit of the following resources affect these parameters:

Resource	Parameter
<code>RLIMIT_NOFILE</code>	<code>OPEN_MAX</code>

When using `getrlimit()`, if a resource limit can be represented correctly in an object of type `rlim_t`, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned is `RLIM_SAVED_MAX`; otherwise, the value returned is `RLIM_SAVED_CUR`.

A limit whose value is greater than `RLIM_INFINITY` is permitted.

The `exec*` family of functions also causes resource limits to be saved.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

The *rlp* argument points to an illegal address.

EINVAL

An invalid resource was specified.

EPERM

The limit specified to [setrlimit\(\)](#) (p. 2899) would have raised the maximum limit value, and the effective user of the calling process isn't the superuser.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

int main( void )
{
    struct rlimit curr_limits;

    if (getrlimit (RLIMIT_NPROC, &curr_limits) == -1) {
        perror ("The call to getrlimit() failed.");
        return EXIT_FAILURE;
    } else {
        printf ("The current maximum number of processes is %d.\n",
                (int) curr_limits.rlim_cur);
        printf ("The hard limit on the number of processes is %d.\n",
                (int) curr_limits.rlim_max);
    }
    return EXIT_SUCCESS;
}
```

Classification:

[getrlimit\(\)](#) is [POSIX 1003.1 XSI](#) (p. 46); [getrlimit64\(\)](#) is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getrusage()

Get information about resource utilization

Synopsis:

```
#include <sys/resource.h>

int getrusage( int who,
               struct rusage * r_usage );
```

Arguments:

who

Which process to get the usage for:

- **RUSAGE_CHILDREN** — get information about resources used by the terminated and waited-for children of the current process. If the child is never waited for (e.g if the parent has **SA_NOCLDWAIT** set, or sets **SIGCHLD** to **SIG_IGN**), the resource information for the child process is discarded and isn't included.
- **RUSAGE_SELF** — get information about resources used by the current process.

r_usage

A pointer to an object of type `struct rusage` in which the function can store the resource information; see below.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *getrusage()* function provides measures of the resources used by the current process or its terminated and waited-for child processes, depending on the value of the *who* argument.

The `rusage` structure is defined as:

```
struct timeval ru_utime;    /* user time used */
struct timeval ru_stime;    /* system time used */
long          ru_maxrss;    /* max resident set size */
```

```

long      ru_ixrss;    /* integral shared memory size */
long      ru_idrss;    /* integral unshared data " */
long      ru_isrss;    /* integral unshared stack " */
long      ru_minflt;   /* page reclaims */
long      ru_majflt;   /* page faults */
long      ru_nswap;    /* swaps */
long      ru_inblock;  /* block input operations */
long      ru_oublock;  /* block output operations */
long      ru_msgsnd;   /* messages sent */
long      ru_msgrcv;   /* messages received */
long      ru_nsignals; /* signals received */
long      ru_nvcsw;    /* voluntary context switches */
long      ru_nivcsw;   /* involuntary " */

```

The members include:

ru_utime

The total amount of time, in seconds and microseconds, spent executing in user mode.

ru_stime

The total amount of time, in seconds and microseconds, spent executing in system mode.

ru_maxrss

The maximum resident set size, given in pages. See the Caveats section, below.

ru_ixrss

Not currently supported.

ru_idrss

An “integral” value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of the process running when a clock tick occurs. The value is given in pages times clock ticks. It doesn't take sharing into account. See the Caveats section, below.

ru_isrss

Not currently supported.

ru_minflt

The number of page faults serviced that didn't require any physical I/O activity. See the Caveats section, below.

ru_majflt

The number of page faults serviced that required physical I/O activity. This could include page ahead operations by the kernel. See the Caveats section, below

ru_nswap

The number of times a process was swapped out of main memory.

ru_inblock

The number of times the file system had to perform input in servicing a *read()* request.

ru_oublock

The number of times the filesystem had to perform output in servicing a *write()* request.

ru_msgsnd

The number of messages sent over sockets.

ru_msgrcv

The number of messages received from sockets.

ru_signals

The number of signals delivered.

ru_nvcsw

The number of times a context switch resulted due to a process's voluntarily giving up the processor before its timeslice was completed (usually to await availability of a resource).

ru_nivcsw

The number of times a context switch resulted due to a higher priority process's becoming runnable or because the current process exceeded its time slice.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EFAULT**

The address specified by the *r_usage* argument isn't in a valid portion of the process's address space.

EINVAL

Invalid *who* parameter.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Only the `timeval` fields of `struct rusage` are supported.

The numbers *ru_inblock* and *ru_oublock* account only for real I/O, and are approximate measures at best. Data supplied by the cache mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.

Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page that isn't in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing *read()* and *write()* functions. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware-detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, major page faults require physical I/O, while minor page faults don't. For example, reclaiming the page from the free list would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in memory. For example, if an address space

faults on some “hot” executable or shared library, a minor page fault results for the address space. Also, anyone doing a *read()* or *write()* to something that's in the page cache gets a minor page fault(s) as well.

There's no way to obtain information about a child process that hasn't yet terminated.

gets()

Get a string of characters from standard input

Synopsis:

```
#include <stdio.h>

char *gets( char *buf );
```

Arguments:

buf

A buffer where the function can store the string.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *gets()* function gets a string of characters from the *stdin* stream, and stores them in the array pointed to by *buf* until end-of-file is encountered or a newline character is read. Any newline character is discarded, and the string is NUL-terminated.



You should use [fgets\(\)](#) (p. 652) instead of *gets()*; *gets()* happily overflows the *buf* array if a newline character isn't read from *stdin* before the end of the array is reached.

The *gets()* function is similar to [fgets\(\)](#) (p. 652), except that *gets()* operates with *stdin*, has no size argument, and replaces a newline character with the NUL character.

Returns:

A pointer to *buf*, or NULL when end-of-file is encountered before reading any characters or a read error occurred ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];

    while( gets( buffer ) != NULL ) {
        puts( buffer );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getservbyname()

Get a service entry, given a name

Synopsis:

```
#include <netdb.h>

struct servent * getservbyname( const char * name,
                                const char * proto );
```

Arguments:

name

The name of the service whose entry you want to find.

proto

NULL, or the protocol for the service.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *getservbyname()* function gets the entry for the given name and protocol from the network services database, `/etc/services`. This function returns a pointer of type [servent](#) (p. 2842), which contains the broken-out fields of a line in the network services database.

The [setservent\(\)](#) (p. 2905) function opens and rewinds the file. If you pass a nonzero *stayopen* argument to *setservent()*, the services database isn't closed after each call to *getservbyname()* or [getservbyport\(\)](#) (p. 954).

The *getservbyname()* and *getservbyport()* functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

Returns:

A valid pointer to a `servent` structure, or NULL if an error occurs.

Files:

/etc/services

Network services database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

getservbyport()

Get a service entry, given a port

Synopsis:

```
#include <netdb.h>

struct servent * getservbyport( int port,
                                const char * proto );
```

Arguments:

port

The port number for the service.

proto

NULL, or the protocol for the service.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *getservbyport()* function gets the entry for the given port from the services database, */etc/services*. This function returns a pointer to a structure of type [servent](#) (p. 2842), which contains the broken-out fields of a line in the network services database.

The [setservent\(\)](#) (p. 2905) function opens and rewinds the file. If you pass a nonzero *stayopen* argument to *setservent()*, the services database isn't closed after each call to [getservbyname\(\)](#) (p. 952) or *getservbyport()*.

The *getservbyport()* function sequentially searches from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

Returns:

A valid pointer to a *servent* structure, or NULL if an error occurs.

Files:

/etc/services

Network services database file.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

getservent()

Read the next line of network services database file

Synopsis:

```
#include <netdb.h>

struct servent * getservent( void );
```

Library:

libsocket
Use the -l socket option to qcc to link against this library.

Description:

The *getservent()* function reads the next line of network services database file, opening the file if necessary. It returns a pointer to a structure of type [servent](#) (p. 2842), which contains the broken-out fields of a line in the network services database, /etc/services.

Returns:

A valid pointer to a `servent` structure, or NULL if an error occurs.

Files:

/etc/services
Network services database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

getsid()

Get the session ID of a process

Synopsis:

```
#include <unistd.h>

pid_t getsid( pid_t pid );
```

Arguments:

pid

The process ID for the process whose session ID you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getsid()* function determines the session ID for the given process ID, *pid*.



In order to get the session ID of a process outside the calling process's session, your process must have the `PROCmgr_AID_GETID` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The session ID, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EPERM

The process specified by *pid* isn't in the same session as the calling process, and the calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

EINVAL

There isn't a process with the given ID.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

getsockname()

Get the name of a socket

Synopsis:

```
#include <sys/socket.h>

int getsockname( int s,
                 struct sockaddr * name,
                 socklen_t * namelen );
```

Arguments:

s

The file descriptor of the socket whose name you want to get.

name

A pointer to a `sockaddr` object where the function can store the socket's name.

namelen

A pointer to a `socklen_t` object that initially indicates the amount of space pointed to by *name*. The function updates *namelen* to contain the actual size of the name (in bytes).

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *getsockname()* function returns the current name for the specified socket.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid descriptor *s*.

EFAULT

The *name* parameter points to memory that isn't in a valid part of the process address space.

ENOBUFS

Insufficient resources were available in the system to perform the operation.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

getsockopt()

Get options associated with a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt( int s,
               int level,
               int optname,
               void * optval,
               socklen_t * optlen );
```

Arguments:

s

The file descriptor of the socket that the option is to be applied to, as returned by [socket\(\)](#) (p. 3075).

level

The protocol layer that the option is to be applied to. In most cases, it's a socket-level option and is indicated by SOL_SOCKET.

optname

The option for the socket file descriptor. For a list of options, see “[Options](#) (p. 963),” below.

optval

A pointer to the value of the option (in most cases, whether the option is to be turned on or off). If no option value is to be returned, *optval* may be NULL.

Most socket-level options use an `int` parameter for *optval*. Others, such as the [SO_LINGER](#) (p. 965), [SO_SNDTIMEO](#) (p. 967), and [SO_RCVTIMEO](#) (p. 966) options, use structures that also let you get data associated with the option.

optlen

A pointer to the length of the value of the option. This argument is a value-result parameter; initialize it to indicate the size of the buffer pointed to by *optval*.

Library:

libsocket

Use the `-l socket` option to `gcc` to link against this library.

Description:

The `getsockopt()` function gets options associated with a socket.

Manipulating socket options

When manipulating a socket option, you must specify the option's name (*optname*) and the level (*level*) at which the option resides.

To manipulate options at the socket-level, specify *level* as `SOL_SOCKET`. When manipulating options any other level, the value that you specify for *level* is represented by the protocol number of the appropriate protocol controlling the option. You can obtain the value in a variety of ways:

- from the “[Options](#) (p. 963)” section below, use the symbolic constant (e.g. `IPPROTO_IP`, `IPPROTO_TCP`) that corresponds to the option
- from `/etc/protocols`, specify the protocol number for the appropriate protocol
- call `getprotobyname()` (p. 924) and pass the appropriate protocol (e.g. `getprotobyname( tcp );`) to retrieve the number of the protocol level.



The latter two ways might not work if you have customized `/etc/protocols`.

The *optname* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The `<sys/socket.h>` header file contains definitions for the socket-level options. Options at other protocol levels vary in format and name.

Since levels (e.g. `SOL_SOCKET`, `IPPROTO_IP` and `IPPROTO_TCP`) and the options within the levels can vary, you need to ensure the proper headers are included for both. For example, when setting `TCP_NODELAY`:



```
int on = 1;
setsockopt(s, IPPROTO_TCP, TCP_NODELAY, &on, sizeof(on));
```

the level `IPPROTO_TCP` is defined in `<netinet/in.h>`, whereas the `TCP_NODELAY` option is defined in `<netinet/tcp.h>`.

Options

This section describes some of the more common options and their corresponding *level*:

IPPROTO_IP:

[IP_HDRINCL](#) (p. 964) · [IP_TOS](#) (p. 964)

SOL_SOCKET:

[SO_BINDTODEVICE](#) (p. 964) · [SO_BROADCAST](#) (p. 964) · [SO_DEBUG](#) (p. 965) · [SO_DONTROUTE](#) (p. 965) · [SO_ERROR](#) (p. 965) · [SO_KEEPALIVE](#) (p. 965) · [SO_LINGER](#) (p. 965) · [SO_OOINLINE](#) (p. 966) · [SO_RCVBUF](#) (p. 966) · [SO_SNDBUF](#) (p. 966) · [SO_RCVLOWAT](#) (p. 966) · [SO_RCVTIMEO](#) (p. 966) · [SO_REUSEADDR](#) (p. 967) · [SO_REUSEPORT](#) (p. 967) · [SO_SNDLOWAT](#) (p. 967) · [SO_SNDTIMEO](#) (p. 967) · [SO_TIMESTAMP](#) (p. 967) · [SO_TYPE](#) (p. 968) · [SO_USELOOPBACK](#) (p. 968)

IPPROTO_TCP

[TCP_KEEPALIVE](#) (p. 968) · [TCP_NODELAY](#) (p. 968)



Except where noted, you can examine the state of the option by calling *getsockopt()*, and set the state by calling *setsockopt()* (p. 2909).

IP_HDRINCL

level: IPPROTO_IP

Get or set the custom IP header that's included with your data. You can use it only for raw sockets. For example:

```
(socket(AF_INET, SOCK_RAW, ...))
```

IP_TOS

level: IPPROTO_IP

Get or set the type-of-service field in the IP header for SOCK_STREAM and SOCK_DGRAM sockets.

SO_BINDTODEVICE

level: SOL_SOCKET

Applies to *setsockopt()* only.

Allow packets to be sent or received on this specified interface only. If the interface specified conflicts with the parameters of *bind()*, or with the routing table, an error or undesired behavior may occur.

This option accepts the *ifreq* structure with the *ifr_name* member set to the interface name (e.g. en0). You can use this option for UDP, TCP, and RAW sockets.

SO_BROADCAST

level: SOL_SOCKET

Enable or disable the permission to transmit broadcast messages. You can use this option only for UDP sockets. For example:

```
(socket(AF_INET, SOCK_DGRAM, ...))
```

“Broadcast” was a privileged operation in earlier versions of the system.

SO_DEBUG

level: SOL_SOCKET

Enable or disable the recording of debug information in the underlying protocol modules.

SO_DONTROUTE

level: SOL_SOCKET

Enable or disable the bypassing of routing tables for outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities. The messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_ERROR

level: SOL_SOCKET

Applies to *getsockopt()* only.

Get any pending error on the socket and clears the error status. You can use it to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SO_KEEPALIVE

level: SOL_SOCKET

Enable or disable the periodic (at least every 2 hours) transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken, and processes that are using the socket are notified via a SIGPIPE signal when they attempt to send data. See “[Keepalive timing](#) (p. 968),” below.

SO_LINGER

level: SOL_SOCKET

Controls the action that's taken when unsent messages are queued on socket when a *close()* is performed.

If it's enabled and the socket promises reliable delivery of data, the system blocks the process on the *close()* attempt until it's able to transmit the data or until it decides it can't deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt()* call when SO_LINGER is requested).

If it's disabled, the system processes the *close()* in a way that lets the process continue as quickly as possible.

The `struct linger` parameter (defined in `<sys/socket.h>`) specifies the desired state of the option in the `l_onoff` field and the linger interval in the `l_linger` field, in seconds. A value of 0 causes a reset on the socket when the application closes the socket.

SO_OOBINLINE

level: SOL_SOCKET

For protocols that support out-of-band data, allows or disallows out-of-band data to be placed in the normal data input queue as received. The data is accessible using the *recv()* or *read()* calls without the MSG_OOB flag. Some protocols always behave as if this option is set.

SO_RCVBUF and SO_SNDBUF

level: SOL_SOCKET

Gets or sets the normal buffer sizes allocated for output (SO_SNDBUF) and input (SO_RCVBUF) buffers. You can increase the buffer size for high-volume connections, or decrease it to limit the possible backlog of incoming data. The system places an absolute limit on these values and defaults them to at least 16K for TCP sockets.

SO_RCVLOWAT

level: SOL_SOCKET

Gets or sets the minimum count for input operations (default is 1). In general, receive calls block until any (nonzero) amount of data is received, and then return with the amount available or the amount requested, whichever is smaller.

If you set the value to be larger than the default, blocking receive calls will wait until they've received the low-water mark value or the requested amount, whichever is smaller. Receive calls may still return less than the low-water mark if: an error occurs, a signal is caught, or if the type of data next in the receive queue differs from that returned.

SO_RCVTIMEO

level: SOL_SOCKET

Gets or sets a timeout value for input operations. It accepts a `struct timeval` parameter (defined in `<sys/time.h>`) with the number of seconds and microseconds used to limit waits for input operations to complete.

In the current implementation, this timer is restarted each time additional data is received by the protocol, so the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or, if no data was received, with the error EWOULDBLOCK.

SO_REUSEADDR*level:* SOL_SOCKET

Enables or disables the reuse of duplicate addresses and port bindings. Indicates that the rules used in validating addresses supplied in a *bind()* call allows/disallows local addresses to be reused.

SO_REUSEPORT*level:* SOL_SOCKET

Enables or disables duplicate address and port bindings. Complete duplicate bindings by multiple processes are allowed when they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. See the reuseport_unicast option of *io-pkt* to see how unicast packets are also received on all sockets bound to the same port.

SO_SNDLOWAT*level:* SOL_SOCKET

Gets or sets the minimum count for output operations. In BSD, this count is typically 2048, but it is a calculated value in QNX Neutrino. If you require a specific SO_SNDLOWAT, you must specify the count. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted (subject to flow control without blocking), but will process no data if flow control doesn't allow the smaller of the low-water mark value or the entire request to be processed.

A *select()* operation that tests the ability to write to a socket returns true only if the low-water mark amount could be processed.

SO_SNDTIMEO*level:* SOL_SOCKET

Gets or sets a timeout value for output operations. It accepts a `struct timeval` parameter (defined in `<sys/time.h>`) that includes the number of seconds and microseconds that are used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error EWOULDBLOCK if data weren't sent.

This timer is restarted each time additional data is delivered to the protocol, implying that the limit applies to output portions ranging in size from the low-water mark to the high-water mark for output. Timeouts are restricted to 32 seconds or under.

SO_TIMESTAMP*level:* SOL_SOCKET

Enables or disables the reception of a timestamp with datagrams. If enabled on a `SOCK_DGRAM` socket, the [recvmsg\(\)](#) (p. 2539) call returns a timestamp corresponding to when the datagram was received. The `msg_control` field in the `msg_hdr` structure points to a buffer that contains a `cmsg_hdr` structure followed by a `struct timeval`. The `cmsg_hdr` fields have the following values:

```
cmsg_len = sizeof(struct cmsg_hdr) + sizeof(struct timeval)
cmsg_level = SOL_SOCKET
cmsg_type = SCM_TIMESTAMP
```

SO_TYPE

level: `SOL_SOCKET`

Applies to *getsockopt()* only.

Gets the type of the socket (e.g. `SOCK_STREAM`). This information is useful for servers that inherit sockets on startup.

SO_USELOOPBACK

level: `SOL_SOCKET`

Enables or disables the sending process to receive its own routing messages.

TCP_KEEPAIVE

level: `IPPROTO_TCP`

Gets or sets the amount of time in seconds between keepalive probes (the default value is 2 hours). It accepts a `struct timeval` parameter with the number of seconds to wait between the keepalive probes. See “[Keepalive timing](#) (p. 968),” below.

TCP_NODELAY

level: `IPPROTO_TCP`

Don't delay sending in order to coalesce packets. Under most circumstances, TCP sends data when it's presented. When outstanding data hasn't yet been acknowledged, TCP gathers small amounts of output to be sent in a single packet once an acknowledgment is received.



For a few clients (such as windowing systems that send a stream of mouse events that receive no replies), this packetization may cause significant delays. Therefore, TCP provides a boolean option, `TCP_NODELAY`, to defeat this algorithm.

Keepalive timing

Use the [SO_KEEPAIVE](#) (p. 965) option to turn on keepalive probes, and the [TCP_KEEPAIVE](#) (p. 968) option to set the amount of idle time before a probe is sent out (the default is 7200 seconds, or two hours).

You can set the number of probes to send out and the interval between probes by using the [sysctl\(\)](#) (p. 3311) function, which modifies or queries the state of the socket manager. The documentation for the `sysctl` utility includes information on identifiers related to `keepcnt` and `keepintvl`. The default number of unsuccessful probes before the socket is considered broken is 8, and the interval between probes defaults to 75 seconds.

Here's an example of turning on a socket keepalive with a 20-second idle time, then up to 3 probes, 5 seconds apart:

```
int mib[4];
int on=1, ival;
struct timeval tval;

mib[0] = CTL_NET;
mib[1] = AF_INET;
mib[2] = IPPROTO_TCP;
mib[3] = TCPCTL_KEEPCNT;
ival = 3; /* Number of keepalive probe attempts
          (default is 8) */
sysctl(mib, 4, NULL, NULL, &ival, sizeof(ival));

mib[0] = CTL_NET;
mib[1] = AF_INET;
mib[2] = IPPROTO_TCP;
mib[3] = TCPCTL_KEEPINTVL;
ival = 10; /* Half seconds between probe attempts;
           default is 150 (75 sec) */
sysctl(mib, 4, NULL, NULL, &ival, sizeof(ival));

memset(&tval, 0, sizeof(tval));
tval.tv_sec = 20; /* Seconds of idle time before probing
                 starts (default is 7200) */
setsockopt(fdi, SOL_SOCKET, SO_KEEPALIVE, (void *) &on,
           sizeof(on));
setsockopt(fdi, IPPROTO_TCP, TCP_KEEPALIVE, (void *) &tval,
           sizeof(tval));
```



Beware that the `sysctl()` calls above make system-wide changes that persist after the process finishes executing.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor `s`.

EDOM

Value was set out of range.

EFAULT

The address pointed to by *optval* isn't in a valid part of the process address space. For *getsockopt()*, this error may also be returned if *optlen* isn't in a valid part of the process address space.

EINVAL

The *optval* argument can't be NULL; *optlen* can't be 0.

ENOPROTOOPT

The option is unknown at the level indicated.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

getspent(), getspent_r()

Get an entry from the shadow password database

Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

struct spwd* getspent( void );

struct spwd* getspent_r( struct spwd* result,
                        char* buffer,
                        int buflen );
```

Arguments:

These arguments apply only to *getspent_r()*:

result

A pointer to a `spwd` structure where the function can store the entry. For more information about this structure, see [putsent\(\)](#) (p. 2459).

buffer

A block of memory that the function can use to allocate storage referenced by the `spwd` structure. You can determine the maximum size needed for this buffer by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETPW_R_SIZE_MAX`.

bufsize

The size of the block that *buffer* points to, in characters.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getspent()* and *getspent_r()* functions return the next entry from the shadow password database. The *getspent()* function uses a static buffer that's overwritten by each call.



The [fgetspent\(\)](#) (p. 654), [getspent\(\)](#), [getspnam\(\)](#) (p. 974), and functions share the same static buffer.

Returns:

The [getspent\(\)](#) function returns a pointer to an object of type `struct spwd` containing the next entry from the shadow password database. When [getspent\(\)](#) is first called, the database is opened, and remains open until either a `NULL` is returned to signify end-of-file, or [endspent\(\)](#) (p. 526) is called.

Errors:

The [getspent\(\)](#) function uses the following functions, and as a result, [errno](#) (p. 539) can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)
- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * This program loops, reading a login name from standard
 * input and checking to see if it is a valid name. If it
 * is not valid, the entire contents of the name in the
 * password database are printed.
 */

int main(int argc, char** argv)
{
    struct spwd* sp;
    char buf[80];

    setpwent( );
    while( gets( buf ) != NULL ) {
        if( ( sp = getspnam( buf ) ) != ( struct spwd * )0 ) {
            printf( "Valid login name is: %s\n", sp->sp_namp );
        } else {
            setspent( );
            while( ( sp=getspent( ) ) != ( struct spwd * )0 )
                printf( "%s\n", sp->sp_namp );
        }
    }
    endspent();
    return( EXIT_SUCCESS );
}
```

Classification:

[Unix](#) (p. 46)

Table 5: *getspent()*

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Table 6: *getspent_r()*

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getspnam(), getspnam_r()

Get information about a user with a given name

Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

struct spwd* getspnam( char* name );

struct spwd* getspnam_r( const char* name,
                        struct spwd* result,
                        char* buffer,
                        size_t bufsize );
```

Arguments:

name

The name of the user.

result

(*getspnam_r()* only) A pointer to a `spwd` structure where the function can store the entry. For more information about this structure, see [putspent\(\)](#) (p. 2459).

buffer

(*getspnam_r()* only) A block of memory that the function can use to allocate storage referenced by the `spwd` structure. You can determine the maximum size needed for this buffer by calling [sysconf\(\)](#) (p. 3308) with an argument of `_SC_GETPW_R_SIZE_MAX`.

bufsize

(*getspnam_r()* only) The size of the block that *buffer* points to, in characters.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getspnam()* and *getspnam_r()* functions allow a process to gain more knowledge about a user *name*. The *getspnam()* function uses a static buffer that's overwritten by each call.



The *fgetspent()* (p. 654), *getspent()* (p. 971), and *getspnam()* functions share the same static buffer.

Returns:

A pointer to an object of type `struct spwd` containing an entry from the group database with a matching *name*, or `NULL` if an error occurred or the function couldn't find a matching entry.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * Print information from the password entry
 * about the user name given as argv[1].
 */

int main( int argc, char** argv )
{
    struct spwd* sp;

    if (argc < 2) {
        printf("%s username \n", argv[0]);
        return(EXIT_FAILURE);
    }

    if( ( sp = getspnam( argv[1] ) ) == (struct spwd*)0) {
        fprintf( stderr, "getspnam: unknown %s\n",
            argv[1] );
        return( EXIT_FAILURE );
    }
    printf( "login name  %s\n", sp->sp_namp );
    printf( "password   %s\n", sp->sp_pwdp );
    return( EXIT_SUCCESS );
}
```

Classification:

Unix (p. 46)

Table 7: *getspnam()*

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No

Safety:	
Thread	No

Table 8: *getspnam_r()*

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getsubopt()

Parse suboptions from a string

Synopsis:

```
#include <stdlib.h>

int getsubopt( char** optionp,
               char* const* tokens,
               char** valuep );
```

Arguments:

optionp

The address of a pointer to the string of options that you want to parse. The function updates this pointer as it parses the options; see below.

tokens

A vector of possible tokens.

valuep

The address of a pointer that the function updates to point to the first character of a value that's associated with an option; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getsubopt()* functions parses suboptions in a flag argument that was initially parsed by *getopt()* (p. 903). These suboptions are separated by commas and may consist of either a single token or a token-value pair separated by an equal sign. Since commas delimit suboptions in the option string, they aren't allowed to be part of the suboption or the value of a suboption. A command that uses this syntax is `mount`, which allows the user to specify mount parameters with the `-o` option as follows:

```
mount -o rw,hard,bg,wsiz=1024 speed:/usr /usr
```

In this example there are four suboptions: `rw`, `hard`, `bg`, and `wsiz`, the last of which has an associated value of 1024.

The *getsubopt()* function takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string, or -1 if there was no match. If the option string at *optionp* contains only one suboption, *getsubopt()* updates *optionp* to point to the null character at the end of the string; otherwise, it isolates the suboption by replacing the comma separator with a null character, and updates *optionp* to point to the start of the next suboption. If the suboption has an associated value, *getsubopt()* updates *valuep* to point to the value's first character. Otherwise, it sets *valuep* to NULL.

The token vector is organized as a series of pointers to null strings. The end of the token vector is identified by a NULL pointer.

When *getsubopt()* returns, if *valuep* isn't NULL, the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when *getsubopt()* fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed to another program.

Returns:

The *getsubopt()* function returns -1 when the token it's scanning isn't in the *tokens* vector. The variable addressed by *valuep* contains a pointer to the first character of the token that wasn't recognized rather than a pointer to a value for that token.

The variable addressed by *optionp* points to the next option to be parsed, or a null character if there are no more options.

Examples:

The following code fragment shows how to process options to the mount(1M) command using *getsubopt()*:

```
#include <stdlib.h>

char *myopts[] = {
#define READONLY 0
    "ro",
#define READWRITE 1
    "rw",
#define WRITESIZE 2
    "wsize",
#define READSIZE 3
    "rsize",
    NULL};

main(argc, argv)
    int argc;
    char **argv;
{
    int sc, c, errflag;
    char *options, *value;
    extern char *optarg;
    extern int optind;
    .
    .
    .
    while((c = getopt(argc, argv, "abf:o:")) != -1) {
```

```

switch (c) {
case 'a': /* process a option */
    break;
case 'b': /* process b option */
    break;
case 'f':
    ofile = optarg;
    break;
case '?':
    errflag++;
    break;
case 'o':
    options = optarg;
    while (*options != '\0') {
        switch(getsubopt(&options, myopts, &value)) {
            case READONLY : /* process ro option */
                break;
            case READWRITE : /* process rw option */
                break;

            case WRITESIZE : /* process wsize option */
                if (value == NULL) {
                    error_no_arg();
                    errflag++;
                } else
                    write_size = atoi(value);
                break;
            case READSIZE : /* process rsize option */
                if (value == NULL) {
                    error_no_arg();
                    errflag++;
                } else
                    read_size = atoi(value);
                break;
            default :
                /* process unknown token */
                error_bad_token(value);
                errflag++;
                break;
        }
    }
    break;
}
}
if (errflag) {
    /* print usage instructions etc. */
}
for (; optind < argc; optind++) {
    /* process remaining arguments */
}
.
.
.
}

```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

During parsing, commas in the option input string are changed to null characters.
White space in tokens or token-value pairs must be protected from the shell by quotes.

gettid()*Get the thread ID***Synopsis:**

```
#include <process.h>

int gettid( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *gettid()* function gets the thread ID for the calling thread.

Returns:

The thread ID.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

int main ( void )
{
    printf( "I'm thread %d\n", gettid() );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

gettimeofday()

Get the current time

Synopsis:

```
#include <sys/time.h>

int gettimeofday( struct timeval * when,
                  void * not_used );
```

Arguments:

when

A pointer to a `timeval` structure where the function can store the time. The `struct timeval` contains the following members:

- `time_t tv_sec` — the number of seconds since the start of the Unix Epoch.
- `suseconds_t tv_usec` — the number of microseconds.

not_used

This pointer must be `NULL` or the behavior of *gettimeofday()* is unspecified. This argument is provided only for backwards compatibility.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *gettimeofday()* function returns the current time in *when* in seconds and microseconds, since the Unix Epoch, 00:00:00 January 1, 1970 Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

Returns:

0 for success, or -1 if an error occurs ([*errno*](#) (p. 539) is set).

Errors:

EFAULT

An error occurred while accessing the *when* buffer.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The *gettimeofday()* function is provided for porting existing code. You shouldn't use it in new code; use [clock_gettime\(\)](#) (p. 312) instead.

getuid()

Get the user ID

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getuid()* function gets the user ID for the calling process.

Returns:

The user ID for the calling process.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "My userid is %d\n", getuid() );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

getutent()

Read the next entry from the user-information file

Synopsis:

```
#include <utmp.h>

struct utmp * getutent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getutent()* function reads in the next entry from a user-information file. If the file isn't already open, *getutent()* opens it. If the function reaches the end of the file, it fails.

Returns:

A pointer to a [utmp](#) (p. 3594) structure for the next entry, or `NULL` if the file couldn't be read or reached the end of file.

Files:

`_PATH_UTMP`

Specifies the user information file.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either [getutid\(\)](#) (p. 987) or [getutline\(\)](#) (p. 989), the routine examines the static structure before performing more I/O. If the contents of the static structure match what it's searching for, the function looks no further. For this reason, to use [getutline\(\)](#) to search for multiple occurrences, zero out the static area after each success, or [getutline\(\)](#) will return the same structure over and over again.

There's one exception to the rule about emptying the structure before further reads are done: the implicit read done by [pututline\(\)](#) (p. 2462) (if it finds that it isn't already at the correct place in the file) doesn't hurt the contents of the static structure returned by the [getutent\(\)](#), [getutid\(\)](#) or [getutline\(\)](#) routines, if you just modified those contents and passed the pointer back to [pututline\(\)](#).

These routines use buffered standard I/O for input, but [pututline\(\)](#) uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

getutid()

Search for an entry in the user-information file

Synopsis:

```
#include <utmp.h>

struct utmp * getutid( struct utmp * id );
```

Arguments:

id

A pointer to a [utmp](#) (p. 3594) structure that you want to find in the user-information file.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getutid()* function searches forward from the current point in the `utmp` file until it finds a matching entry:

- If *id->ut_type* is one of `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, or `NEW_TIME`, the function looks for an entry with the same *ut_type*.
- If *id->ut_type* is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS`, *getutid()* looks for the first entry entry with the same *ut_type* and a *ut_id* field that matches *id->ut_id*.

If *getutid()* reaches the end of the file without finding a match, the search fails.

Returns:

A pointer to the [utmp](#) (p. 3594) structure for the matching entry, or `NULL` if it couldn't be found.

Files:

_PATH_UTMP

Specifies the user information file.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either *getutid()* or *getutline()* (p. 989), the routine examines the static structure before performing more I/O. If the contents of the static structure match what it's searching for, the function looks no further. For this reason, to use *getutline()* to search for multiple occurrences, zero out the static area after each success, or *getutline()* will return the same structure over and over again.

There's one exception to the rule about emptying the structure before further reads are done: the implicit read done by *pututline()* (p. 2462) (if it finds that it isn't already at the correct place in the file) doesn't hurt the contents of the static structure returned by the *getutent()* (p. 985), *getutid()* or *getutline()* routines, if the user has just modified those contents and passed the pointer back to *pututline()*.

These routines use buffered standard I/O for input, but *pututline()* uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

getutline()

Get an entry from the user-information file

Synopsis:

```
#include <utmp.h>

struct utmp * getutline( struct utmp * line );
```

Arguments:

line

A pointer to a [utmp](#) (p. 3594) structure that you want to find in the user-information file.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *getutline()* function searches forward from the current point in the `utmp` file until it finds an entry of the type `LOGIN_PROCESS` or a `ut_line` string that matches `line->ut_line`. If the function reaches the end of the file is reached without finding a match, the function fails.

Returns:

A pointer to the [utmp](#) (p. 3594) structure for the entry found, or `NULL` if the search failed.

Files:

`_PATH_UTMP`

Specifies the user information file.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either *getutid()* (p. 987) or *getutline()*, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it's searching for, the function looks no further. For this reason, to use *getutline()* to search for multiple occurrences, zero out the static area after each success, or *getutline()* will return the same structure over and over again.

There's one exception to the rule about emptying the structure before further reads are done: the implicit read done by *pututline()* (p. 2462) (if it finds that it isn't already at the correct place in the file) doesn't hurt the contents of the static structure returned by the *getutent()* (p. 985), *getutid()* or *getutline()* routines, if the user has just modified those contents and passed the pointer back to *pututline()*.

These routines use buffered standard I/O for input, but *pututline()* uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

getw()

Get a word from a stream

Synopsis:

```
#include <stdio.h>

int getw( FILE* stream );
```

Arguments:

stream

The stream that you want to read a word from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getw()` function returns the next word (i.e. integer) from the named input stream. This function increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer, and varies from machine to machine. The `getw()` function assumes no special alignment in the file.

Returns:

The next word, or the constant `EOF` at the end-of-file or on an error; it sets the `EOF` or error indicator of the stream.



Use `feof()` (p. 637) or `ferror()` (p. 639) to distinguish an end-of-file condition from an error.

Errors:

EOVERFLOW

The file is a regular file, and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Because of possible differences in word length and byte ordering, files written using *putw()* (p. 2464) are implementation-dependent, and might not be read correctly using *getw()* on a different processor.

getwc()

Read a wide character from a stream

Synopsis:

```
#include <wchar.h>

wint_t getwc( FILE * fp );
```

Arguments:

fp

The stream from which you want to read a wide character.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getwc()` function reads the next wide character from the specified stream.

Returns:

The next character from the stream, cast as `(wint_t)(wchar_t)`, or `WEOF` if end-of-file has been reached or if an error occurs ([errno](#) (p. 539) is set).



Use [feof\(\)](#) (p. 637) or [ferror\(\)](#) (p. 639) to distinguish an end-of-file condition from an error.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *fp* and would have been blocked by this operation.

EBADF

The *fp* stream isn't valid for reading.

EINTR

A signal terminated the read operation; no data was transferred.

EIO

Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.

Eoverflow

Cannot read at or beyond the offset maximum for this stream.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getwchar()

Read a character from a stream

Synopsis:

```
#include <wchar.h>

wint_t getwchar( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *getwchar()* function reads the next wide character from *stdin*.

Returns:

The next character from *stdin*, cast as `(wint_t)(wchar_t)`, or `WEOF` if the end-of-file has been reached or if an error occurs (*errno* (p. 539) is set).



Use *feof()* (p. 637) or *ferror()* (p. 639) to distinguish an end-of-file condition from an error.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *stdin* and would have been blocked by this operation.

EINTR

A signal terminated the read operation; no data was transferred.

EIO

Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.

E_OVERFLOW

Cannot read at or beyond the offset maximum for this stream.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

getwd()

Get current working directory pathname

Synopsis:

```
#include <unistd.h>

char* getwd( char *path_name );
```

Arguments:

path_name

A buffer where the function can store the current working directory.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getwd()` function determines the absolute pathname of the current working directory of the calling process, and copies that pathname into the array pointed to by the `path_name` argument.

If the length of the pathname of the current working directory is greater than $(\{\text{PATH\_MAX}\} + 1)$ including the null byte, `getwd()` fails and returns a null pointer.



For portability, use `getcwd()` (p. 830) instead of `getwd()`.

Returns:

A pointer to the string containing the absolute pathname of the current working directory. On error, `getwd()` returns a null pointer and the contents of the array pointed to by `path_name` are undefined.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

glob()

Find paths matching a pattern

Synopsis:

```
#include <glob.h>

int glob( const char* pattern,
          int flags,
          int (*errfunc)( const char* epath,
                          int error ),
          glob_t* pglob );
```

Arguments:

pattern

The pattern you want to match. This can include these wildcard characters:

- `*` matches any string, of any length
- `?` matches any single character
- `[chars]` matches any of the characters found in the string *chars*.

flags

Flags that affect the search; see below.

errfunc

A pointer to a function that *glob()* calls when it encounters a directory that it can't open or read. For more information, see below.

pglob

A pointer to a `glob_t` structure where *glob()* can store the paths found. This structure contains at least the following members:

- `size_t gl_pathc` — the number of pathnames matched by *pattern*.
- `char** gl_pathv` — a NULL-terminated array of pointers to the pathnames matched by *pattern*.
- `size_t gl_offs` — the number of pointers to reserve at the beginning of *gl_pathv*.

You must create the `glob_t` structure before calling *glob()*. The *glob()* function allocates storage as needed for the *gl_pathv* array. Use [*globfree\(\)*](#) (p. 1004) to free this space.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `glob()` function finds pathnames matching the given pattern.

In order to have access to a pathname, `glob()` must have search permission on every component of the path except the last, and read permission on each directory of every filename component of *pattern* that contains any of the special characters (`*`, `?`, `[` and `]`).

The *errfunc* argument is a pointer to an error-handler function with this prototype:

```
int errfunc( const char* epath, int error );
```

The *errfunc* function is called when `glob()` encounters a directory that it can't open or read. The arguments are:

epath

A pointer to the path that failed.

error

The value of [errno](#) (p. 539) from the failure. The *error* argument can be set to any of the values returned by [opendir\(\)](#) (p. 1913), [readdir\(\)](#) (p. 2514), or [stat\(\)](#) (p. 3164).

The *errfunc* function should return 0 if `glob()` should continue, or a nonzero value if `glob()` should stop searching.

You can set *errfunc* to `NULL` to ignore these types of errors.

The *flags* argument can be set to any combination of the following bits:

GLOB_APPEND

Append found pathnames to the ones from a previous call from `glob()`.

GLOB_DOOFFS

Use the value in `pglob->gl_offs` to specify how many `NULL` pointers to add at the beginning of `pglob->pathv`. After the call to `glob()`, `pglob->pathv` will contain `pglob->gl_offs` `NULL` pointers, followed by `pglob->gl_pathc`

pathnames, followed by a NULL pointer. This can be useful if you're building a command to be applied to the matched files.

GLOB_ERR

Cause *glob()* to return when it encounters a directory that it can't open or read. Otherwise, *glob()* will continue to find matches.

GLOB_MARK

Append a slash to each matching pathname that's a directory.

GLOB_NOCHECK

If *pattern* doesn't match any path names, return only the contents of *pattern*.

GLOB_NOESCAPE

Disable backslash escapes in *pattern*.

GLOB_NOSORT

Don't sort the returned pathnames; their order will be unspecified. The default is to sort the pathnames.

The following flags are BSD extensions:

GLOB_PERIOD

Allow metacharacters to match leading periods.

GLOB_MAGCHAR

Pattern had globbing characters.

GLOB_ALTDIRFUNC

Use alternately specified directory functions.

GLOB_BRACE

Expand braces the way that the C shell does.

GLOB_NOMAGIC

Similar to GLOB_NOCHECK without magic characters (csh).

GLOB_TILDE

Expand tilde names from the passwd file.

GLOB_NO_DOTDIRS

Make . and . . vanish from wildcards.

GLOB_LIMIT

Limit memory used by matches to ARG_MAX.

Returns:

Zero for success, or an error value.

Errors:**GLOB_ABEND**

The scan was stopped because GLOB_ERR was set, or the *errfunc* function returned nonzero.

GLOB_NOMATCH

The value of *pattern* doesn't match any existing pathname, and GLOB_NOCHECK wasn't set in *flags*.

GLOB_NOSPACE

Unable to allocate memory to store the matched paths.

Examples:

This simple example attempts to find all of the .c files in the current directory and print them in the order the filesystem found them.

```
#include <unistd.h>
#include <stdio.h>
#include <glob.h>

int main( void )
{
    glob_t paths;
    int retval;

    paths.gl_pathc = 0;
    paths.gl_pathv = NULL;
    paths.gl_offs = 0;

    retval = glob( "*.c", GLOB_NOCHECK | GLOB_NOSORT,
                  NULL, &paths );
    if( retval == 0 ) {
        int idx;

        for( idx = 0; idx < paths.gl_pathc; idx++ ) {
            printf( "[%d]: %s\n", idx,
                  paths.gl_pathv[idx] );
        }

        globfree( &paths );
    } else {
        puts( "glob() failed" );
    }

    return 0;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Don't change the values in *pglob* between calling *glob()* and [globfree\(\)](#) (p. 1004).

globfree()

Free storage allocated by a call to glob()

Synopsis:

```
#include <glob.h>

void globfree( glob_t* pglob );
```

Arguments:

pglob

A pointer to a `glob_t` structure that you passed to [glob\(\)](#) (p. 999).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *globfree()* function frees the storage allocated by a call to *glob()*.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Don't change the values in *pglob* between calling *glob()* and *globfree()*.

gmtime()

Convert calendar time to a broken-down time

Synopsis:

```
#include <time.h>

struct tm* gmtime( const time_t* timer );
```

Arguments:

timer

A pointer to a `time_t` structure that contains the time that you want to convert.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `gmtime()` function converts the calendar time pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time or GMT).



The `gmtime()` function places the converted time in a static structure that's reused each time you call `gmtime()` or `localtime()` (p. 1531). Calling `asctime()` (p. 155) or `ctime()` (p. 389) could also change the data in this static buffer.

In a multithreaded application, use `gmtime_r()` (p. 1007).

You typically use the `date` command to set the computer's internal clock using Coordinated Universal Time (UTC). Use the **TZ** environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Returns:

A pointer to the static `tm` (p. 3476) structure that contains the broken-down time.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

gmtime_r()

Convert calendar time to a broken-down time

Synopsis:

```
#include <time.h>

struct tm* gmtime_r( const time_t* timer,
                     struct tm* result );
```

Arguments:

timer

A pointer to a `time_t` structure that contains the time that you want to convert.

result

A pointer to a [tm](#) (p. 3476) structure where the function can store the broken-down time.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *gmtime_r()* function converts the calendar time pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time or GMT) and stores it in the [tm](#) (p. 3476) structure pointed to by *result*.

You typically use the `date` command to set the computer's internal clock using Coordinated Universal Time (UTC). Use the ***TZ*** environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Returns:

A pointer to the [tm](#) (p. 3476) structure containing the broken-down time.

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

grantpt()

Grant access to a slave pseudo-terminal device

Synopsis:

```
#include <stdlib.h>

int grantpt( int fildes );
```

Arguments:

fildes

The file descriptor of a master pseudo-terminal device.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *grantpt()* function changes the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counterpart. The *fildes* argument is a file descriptor that refers to a master pseudo-terminal device. The user and group IDs of the slave are set to the real user and group IDs of the calling process. The permission mode of the slave pseudo-terminal is set to readable and writable by the owner, and writable by the group.

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The *fildes* argument isn't a valid open file descriptor.

EINVAL

The *fildev* argument isn't associated with a master pseudo-terminal device.

EACCES

The corresponding slave pseudo-terminal device couldn't be accessed.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 10

H

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “H”.

h_errno

Host error variable

Synopsis:

```
#include <netdb.h>

extern int h_errno;
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *h_errno* variable can be set by any one of the following functions:

- [gethostbyaddr\(\)](#) (p. 866)
- [gethostbyaddr_r\(\)](#) (p. 869)
- [gethostbyname\(\)](#) (p. 872)
- [gethostbyname2\(\)](#) (p. 872)
- [gethostbyname_r\(\)](#) (p. 875)
- [res_query\(\)](#) (p. 2569)
- [res_search\(\)](#) (p. 2575)

It can be set to any one of the following:

HOST_NOT_FOUND

Authoritative answer: Unknown host.

NETDB_INTERNAL

You specified an invalid address family when calling [gethostbyname2\(\)](#) (p. 872).

NO_DATA

Valid name, no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn't a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

NO_RECOVERY

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

TRY_AGAIN

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn't receive a response from an authoritative server. A retry at some later time may succeed.

Classification:

POSIX 1003.1 OBS (p. 46)

hcreate()

Create a hash search table

Synopsis:

```
#include <search.h>

int hcreate( size_t nel );
```

Arguments:

nel

An estimate of the maximum number of entries that the table will contain. The algorithm might adjust this number upward in order to obtain certain mathematically favorable circumstances.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *hcreate()* function allocates space for the hash search table. You must call this function before using [hsearch\(\)](#) (p. 1021).

The *hsearch()* and *hcreate()* functions use [malloc\(\)](#) (p. 1569) to allocate space.

Only one hash search table may be active at any given time. You can destroy the table by calling [hdestroy\(\)](#) (p. 1016).

See also *The Art of Computer Programming, Volume 3, Sorting and Searching* by Donald E. Knuth, published by Addison-Wesley Publishing Company, 1973.

Returns:

0 if there isn't enough space available to allocate the table.

Examples:

See [hsearch\(\)](#) (p. 1021).

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

hdestroy()

Destroy the hash search table

Synopsis:

```
#include <search.h>

void hdestroy( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *hdestroy()* function destroys the hash search table that was created by [hcreate\(\)](#) (p. 1014) and used by [hsearch\(\)](#) (p. 1021). Only one hash search table may be active at any given time.

Examples:

See [hsearch\(\)](#) (p. 1021).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

herror()

Print the message associated with the value of `h_errno` to standard error

Synopsis:

```
#include <netdb.h>

void herror( const char* prefix );
```

Arguments:

prefix

NULL, or a string that you want to print before the error message.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *herror()* function prints the message corresponding to the error number contained in *h_errno* (p. 1012) to *stderr* (p. 3175). The following functions can set *h_errno*:

- [gethostbyaddr\(\)](#) (p. 866)
- [gethostbyaddr_r\(\)](#) (p. 869)
- [gethostbyname\(\)](#) (p. 872)
- [gethostbyname2\(\)](#) (p. 872)
- [gethostbyname_r\(\)](#) (p. 875)
- [res_query\(\)](#) (p. 2569)
- [res_search\(\)](#) (p. 2575)

If the *prefix* string is non-NULL, it's printed, followed by a colon and a space. The error message is printed with a trailing newline. One of the following messages could be printed:

HOST_NOT_FOUND

Authoritative answer: Unknown host.

NETDB_INTERNAL

You specified an invalid address family when calling [gethostbyname2\(\)](#) (p. 872).

NO_DATA

Valid name, no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn't a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

NO_RECOVERY

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

TRY_AGAIN

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn't receive a response from an authoritative server. A retry at some later time may succeed.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

hostent

Structure that describes an Internet host

Synopsis:

```
#include <netdb.h>

struct hostent {
    char *   h_name;
    char **  h_aliases;
    int      h_addrtype;
    int      h_length;
    char **  h_addr_list;
};

#define h_addr  h_addr_list[0]
```

Description:

This structure describes an Internet host. It contains either the information obtained from a name server, or broken-out fields from a line in `/etc/hosts`.

The members of this structure are:

h_name

The official name of the host.

h_aliases

A zero-terminated array of alternate names for the host.

h_addrtype

The type of address being returned; currently always `AF_INET`.

h_length

The length of the address, in bytes.

h_addr_list

A zero-terminated array of network addresses for the host. Host addresses are returned in network byte order.

A `#define` statement is used to define the following:

h_addr

The first address in *h_addr_list*. This is for backward compatibility.

Classification:

POSIX 1003.1 (p. 46)

hsearch()

Search the hash search table

Synopsis:

```
#include <search.h>

ENTRY* hsearch ( ENTRY item,
                 ACTION action );
```

Arguments:

item

A structure of type ENTRY, defined in <search.h>, that contains:

- char **key* — a pointer to the comparison key.
- void **data* — a pointer to any other data to be associated with the key.

action

A member of an enumeration type ACTION, also defined in <search.h>, indicating what to do with the entry if it isn't in the table:

- ENTER — insert the entry in the table at the appropriate point. If the item is a duplicate of an existing item, the new item isn't added, and *hsearch()* returns a pointer to the existing one.
- FIND — don't add the entry. If the item can't be found, *hsearch()* returns NULL.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *hsearch()* function is a hash-table search routine generalized from Knuth (6.4) Algorithm D. Before using this function, you must call [hcreate\(\)](#) (p. 1014) to create the hash table.

The `hsearch()` function returns a pointer into a hash table indicating the location at which an entry can be found. This function uses `strcmp()` (p. 3186) as the comparison function.

The `hsearch()` and `hcreate()` functions use `malloc()` (p. 1569) to allocate space.

Only one hash search table may be active at any given time. You can destroy the table by calling `hdestroy()` (p. 1016).

See also *The Art of Computer Programming, Volume 3, Sorting and Searching* by Donald E. Knuth, published by Addison-Wesley Publishing Company, 1973.

Returns:

A pointer to the item found, or NULL if either the action is FIND and the item wasn't found, or the action is ENTER and the table is full.

Examples:

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings, finds the matching entry in the hash table and prints it.

```
#include <stdio.h>
#include <search.h>
#include <string.h>
#include <stdlib.h>
struct info {
    int age, room;          /* this is the info stored in table */
                          /* other than the key */
};
#define NUM_EMPL    5000 /* # of elements in search table */
main( )
{
    /* space to store strings */
    char string_space[NUM_EMPL*20];
    /* space to store employee info */
    struct info info_space[NUM_EMPL];
    /* next avail space in string_space */
    char *str_ptr = string_space;
    /* next avail space in info_space */
    struct info *info_ptr = info_space;
    ENTRY item, *found_item;
    /* name to look for in table */
    char name_to_find[30];
    int i = 0;

    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
        &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* put info in structure, and structure in item */
        item.key = str_ptr;
        item.data = (void *)info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        /* put item into table */
        (void) hsearch(item, ENTER);
    }

    /* access table */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {
            /* if item is in the table */
            (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
```

```
        ((struct info *)found_item->data)->room);
    } else {
        (void)printf("no such employee %s\n",
                     name_to_find);
    }
}
hdestroy();
return 0;
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

hstrerror()

Get an error message string associated with the error return status

Synopsis:

```
#include <netdb.h>

const char* hstrerror( int err );
```

Arguments:

err

The error code that you want to get the message for. For more information, see [h_errno](#) (p. 1012).

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *hstrerror()* function gets an error message string associated with the error return status from network host-related functions.

Network host-related functions such as the following can return the error status:

- [gethostbyaddr\(\)](#) (p. 866), [gethostbyaddr_r\(\)](#) (p. 869)
- [gethostbyname\(\)](#) (p. 872), [gethostbyname_r\(\)](#) (p. 875)
- [res_query\(\)](#) (p. 2569)
- [res_search\(\)](#) (p. 2575)

You can check the external integer *h_errno* to see whether this is a temporary failure or an invalid or unknown host.

Returns:

A pointer to the message string affiliated with an error number.



Don't modify the message string.

Classification:[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

htonl()

Convert a 32-bit value from host-byte order to network-byte order

Synopsis:

```
#include <arpa/inet.h>

uint32_t htonl( uint32_t hostlong );
```

Arguments:

hostlong

The value that you want to convert.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *htonl()* function converts a 32-bit value from host-byte order to network-byte order. You typically use this routine in conjunction with the internet addresses and ports that [gethostbyname\(\)](#) (p. 872) and [getservent\(\)](#) (p. 956) return.

Returns:

The value in network-byte order.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

htons()

Convert a 16-bit value from host-byte order to network-byte order

Synopsis:

```
#include <arpa/inet.h>

uint16_t htons( uint16_t hostshort );
```

Arguments:

hostshort

The value that you want to convert.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *htons()* function converts a 16-bit value from host-byte order to network-byte order.

You typically use this routine in conjunction with the internet addresses and ports that [gethostbyname\(\)](#) (p. 872) and [getservent\(\)](#) (p. 956) return.

Returns:

The value in network-byte order.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

hwi_find_item()

Find an item in the hwinfo structure

Synopsis:

```
#include <hw/sysinfo.h>

unsigned hwi_find_item( unsigned start,
                      ... );
```

Arguments:

start

Where to start the search for the given item.

For the initial call, set this argument to `HWI_NULL_OFF`. If the item found isn't the one that you want, pass the return value from the first call to *hwi_find_item()* as the *start* parameter of the next call. This makes the search pick up where it left off. You can repeat this process as many times as required (the return value from the second call going into the *start* parameter of the third, etc).

*char **

A sequence of names for identifying the item being searched.

Terminate the sequence with a NULL pointer. The last string before the NULL is the bottom-level item name that you're looking for, the string in front of that is the name of the item that owns the bottom-level item, etc.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *hwi_find_item()* function finds an item in the *hwinfo* structure of the system page.

Returns:

The offset of the item requested, or HWI_NULL_OFF if the item wasn't found.

Examples:

Find the first occurrence of an item called “foobar”:

```
item_off = hwi_find_item(HWI_NULL_OFF, "foobar", NULL);
```

Find the first occurrence of an item called “foobar” that's owned by “sam”:

```
item_off = hwi_find_item(HWI_NULL_OFF, "sam", "foobar", NULL);
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

hwi_find_tag()

Find a tag in the hwinfo structure

Synopsis:

```
#include <hw/sysinfo.h>

unsigned hwi_find_tag( unsigned start,
                      int curr_item,
                      const char * tagname );
```

Arguments:

start

Where to start to search for the given item.

For the initial call, set this argument to `HWI_NULL_OFF`. If the item found isn't the one that you want, pass the return value from the first call to *hwi_find_tag()* as the *start* parameter of the next call. This makes the search pick up where it left off. You can repeat this process as many times as required (the return value from the second call going into the *start* parameter of the third, etc).

curr_item

If this argument is nonzero, the search stops at the end of the current item (i.e. the one that *start* points to). If *curr_item* is zero, the search continues until the end of the section.

tagname

The name of tag to search for.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *hwi_find_tag()* function finds the tag named *tagname*.

Returns:

The offset of the tag, or `HWI_NULL_OFF` if the tag wasn't found.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

hwi_next_item()

Find the next item in the hwinfo structure

Synopsis:

```
#include <hw/sysinfo.h>

unsigned hwi_next_item( unsigned off );
```

Arguments:

off

The offset, in bytes from the start of the *hwinfo* section, to start from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *hwi_next_item()* function finds the next item in the *hwinfo* structure of the system page.

Returns:

The offset of the next item, or `HWI_NULL_OFF` if there isn't another item.

Examples:

This program produces a summary of the information that the `pidin syspage=hwinfo` command displays:

```
#include <stdio.h>
#include <stdlib.h>
#include <hw/sysinfo.h>
#include <sys/syspage.h>

int main ( void )
{
    unsigned item_offset = 0, tag_offset = 0;
    struct hwi_item *curr_item, *curr_tag;
    int first_tag;
```

```

printf ("Item          Offset  Owner  Tags\n");
printf ("-----  -----  ----  ----\n");

while (item_offset != HWI_NULL_OFF)
{
    /* Print the type, name, and offset of the current item */
    curr_item = hwi_off2tag (item_offset);
    printf ("%6s %-11s %3d",
            &SYSPAGE_ENTRY(strings)->data[curr_item->prefix.name],
            &SYSPAGE_ENTRY(strings)->data[curr_item->itemname],
            item_offset);

    /* Print the offset of the owner (if any) */
    if (curr_item->owner == HWI_NULL_OFF)
    {
        printf ("      ");
    } else {
        printf ("    %3d", curr_item->owner);
    }

    /* List the tags in this item */
    first_tag = 1;
    tag_offset = hwi_next_tag (item_offset, 1);

    while (tag_offset != HWI_NULL_OFF)
    {
        printf ("%s", first_tag ? "      " : ", ");
        first_tag = 0;

        curr_tag = hwi_off2tag (tag_offset);
        printf ("%s",
                &SYSPAGE_ENTRY(strings)->data[curr_tag->prefix.name]);

        /* Get the offset of the next tag in the current item */
        tag_offset = hwi_next_tag (tag_offset, 1);
    }

    printf ("\n");

    /* Get the offset of the next item */
    item_offset = hwi_next_item (item_offset);
}
return (EXIT_SUCCESS);
}

```

The output from this program looks something like this:

Item		Offset	Owner	Tags
----		-----	-----	----
Group	hw	0		
Bus	unknown	12	0	
Group	rtc	24	12	
Device	mc146818	36	24	pad, location
Bus	pci	80	0	
Device	bios	92	80	
Bus	isa	108	0	
Group	disk	120	108	diskgeometry
Group	misc	148	108	
Device	pckbd	160	148	irq, location
Group	serial	208	108	
Device	8250	220	208	irq, pad, location, irq, location
Group	parallel	304	108	
Device	centronics	316	304	pad, location
Group	misc	360	12	
Device	in-use	372	360	pad, location, location, location, location, irq, irq

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

hwi_next_tag()

Find the next tag in the hwinfo structure

Synopsis:

```
#include <hw/sysinfo.h>

unsigned hwi_next_tag( unsigned off,
                      int  curr_item );
```

Arguments:

off

The offset, in bytes from the start of the *hwinfo* section, to start from.

curr_item

If this argument is nonzero, the search stops at the end of the current item (i.e. the one that *start* points to). If *curr_item* is zero, the search continues until the end of the section.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *hwi_next_tag()* function finds the next tag after the given offset.

Returns:

The offset of the tag, or `HWI_NULL_OFF` if there isn't another tag.

Examples:

See [hwi_next_item\(\)](#) (p. 1032).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

hwi_off2tag()

Return a pointer to the start of a tag in the hwinfo area of the system page

Synopsis:

```
#include <hw/sysinfo.h>

void * hwi_off2tag( unsigned offset );
```

Arguments:

offset

The offset, in bytes from the start of the *hwinfo* section, of a tag.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *hwi_off2tag()* function returns a pointer to the start of the tag, given an offset.

Returns:

A pointer to the start of the tag.

Examples:

See [hwi_next_item\(\)](#) (p. 1032).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

hwi_tag2off()

Return the offset from the start of the hwinfo area of the system page

Synopsis:

```
#include <hw/sysinfo.h>

unsigned hwi_tag2off( void *tag );
```

Arguments:

tag

A pointer to a tag in the *hwinfo* area of the system page.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

Given a pointer to the start of a tag, the *hwi_tag2off()* function returns the offset, in bytes, from the beginning of the start of the *hwinfo* section.

Returns:

The offset of the tag, in bytes.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

hypot(), hypotf()

Calculate the length of the hypotenuse for a right-angled triangle

Synopsis:

```
#include <math.h>

double hypot( double x,
              double y );

float hypotf( float x,
              float y );
```

Arguments:

x, y

The lengths of the sides that are adjacent to the right angle.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the length of the hypotenuse for a right triangle whose sides are *x* and *y* adjacent to the right angle. The calculation is equivalent to:

```
length = sqrt( x*x + y*y );
```

Returns:

The length of the hypotenuse.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
```

```
    printf( "%f\n", hypot( 3.0, 4.0 ) );  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
5.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 11

I

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>ICMP6_FILTER_SETPASSALL()</i>	ICMP6 (p. 1047)
<i>ICMP6_FILTER_SETBLOCKALL()</i>	ICMP6 (p. 1047)
<i>ICMP6_FILTER_SETPASS()</i>	ICMP6 (p. 1047)
<i>ICMP6_FILTER_SETBLOCK()</i>	ICMP6 (p. 1047)
<i>ICMP6_FILTER_WILLPASS()</i>	ICMP6 (p. 1047)
<i>ICMP6_FILTER_WILLBLOCK()</i>	ICMP6 (p. 1047)
<i>_IO_SET_CONNECT_RET()</i>	iofunc_open() (p. 1321)
<i>_IO_SET_PATHCONF_VALUE()</i>	iofunc_pathconf() (p. 1332)
<i>_IO_SET_READ_NBYTES()</i>	iofunc_read_verify() (p. 1339)
<i>_IO_SET_WRITE_NBYTES()</i>	iofunc_write_verify() (p. 1382)
<i>IOFUNC_NOTIFY_DISARM()</i>	iofunc_notify() (p. 1300)
<i>IOFUNC_NOTIFY_INIT()</i>	iofunc_notify() (p. 1300)
<i>IOFUNC_NOTIFY_INPUT_CHECK()</i>	iofunc_notify_trigger() (p. 1308)
<i>IOFUNC_NOTIFY_OBAND_CHECK()</i>	iofunc_notify_trigger() (p. 1308)
<i>IOFUNC_NOTIFY_OUTPUT_CHECK()</i>	iofunc_notify_trigger() (p. 1308)

The following have been deprecated:

Instead of:	Use:
<i>intr_capture()</i>	InterruptAttach() (p. 1147)
<i>intr_lock()</i>	InterruptMask() (p. 1176)
<i>intr_release()</i>	InterruptDetach() (p. 1162)

Instead of:	Use:
<i>intr_timed_wait()</i>	<i>InterruptWait()</i> (p. 1183) and <i>Timertimeout()</i> (p. 3464)
<i>intr_unlock()</i>	<i>InterruptUnmask()</i> (p. 1181)
<i>intr_wait()</i>	<i>InterruptWait()</i> (p. 1183)
<i>iofunc_client_info()</i>	<i>iofunc_client_info_ext()</i> (p. 1246)
<i>isinff()</i>	<i>isinf()</i> (p. 1441)
<i>isnanf()</i>	<i>isnan()</i> (p. 1445)

This chapter describes the API components whose names begin with “I”.

ICMP

Internet Control Message Protocol

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_RAW,
            proto );
```

Description:

ICMP is the error- and control-message protocol used by IP and the Internet protocol family. The protocol may be accessed through a “raw socket” for network monitoring and diagnostic functions.

To get the *proto* parameter to [socket\(\)](#) (p. 3075) that's used to create an ICMP socket, call [getprotobyname\(\)](#) (p. 924). You normally use ICMP sockets, which are connectionless, with [sendto\(\)](#) (p. 2837) and [recvfrom\(\)](#) (p. 2536), although you can also use [connect\(\)](#) (p. 348) to fix the destination for future packets (in which case you can use the [read\(\)](#) (p. 2502) or [recv\(\)](#) (p. 2533), and [write\(\)](#) (p. 3737) or [send\(\)](#) (p. 2829) system calls).

Outgoing packets automatically have an IP header prepended to them that's based on the destination address. Incoming packets are received with the IP header and IP options intact.

Based on:

RFC 792

Returns:

A descriptor referencing the socket, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EADDRNOTAVAIL

Tried to create a socket with a network address for which no network interface exists.

EISCONN

Tried to establish a connection on a socket that already has one, or tried to send a datagram with the destination address specified but the socket is already connected.

ENOBUFFS

The system ran out of memory for an internal data structure.

ENOTCONN

Tried to send a datagram, but no destination address was specified, and the socket hasn't been connected.

ICMP6

Internet Control Message Protocol for IP6

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/icmp6.h>

int socket( AF_INET6,
            SOCK_RAW,
            proto);
```

Description:

ICMP6 is the error and control message protocol that IP6 and the Internet Protocol family use. It may be accessed through a “raw socket” for network monitoring and diagnostic functions. Use the [getprotobyname\(\)](#) (p. 924) function to obtain the *proto* parameter to the [socket\(\)](#) (p. 3075) function, or simply pass IPPROTO_ICMPV6.

ICMPv6 sockets are connectionless, and are normally used with the [sendto\(\)](#) (p. 2837) and [recvfrom\(\)](#) (p. 2536) functions. You may also use the [connect\(\)](#) (p. 348) function to fix the destination for future packets (in which case, you may also use the [read\(\)](#) (p. 2502) or [recv\(\)](#) (p. 2533) functions and the [write\(\)](#) (p. 3737) or [send\(\)](#) (p. 2829) system calls).

Outgoing packets automatically have an [IP6](#) (p. 1412) header prepended to them (based on the destination address). The ICMP6 pseudo header checksum field (*icmp6_cksum*, found in the *icmp6_hdr* structure in *<netinet/icmp6.h>*) is filled automatically by the socket manager. Incoming packets are received without the IP6 header or extension headers.



This behavior is opposite from both IPv4 raw sockets and ICMPv4 sockets.

ICMP6 type/code filter

Each ICMP6 raw socket has an associated filter whose data type is defined as *struct icmp6_filter*. This structure, along with the macros and constants defined below are defined in the *<netinet/icmp6.h>* header.

You can get and set the current filter by calling [getsockopt\(\)](#) (p. 962) and [setsockopt\(\)](#) (p. 2909) with a level of IPPROTO_ICMPV6 and an option name of ICMP6_FILTER.

The following macros operate on an *icmp6_filter* structure. If the first argument is an integer, it represents an ICMP6 message type, with a value between 0 and 255. The pointer arguments are pointers to the filters that are either set or examined, depending on the macro:

ICMP6_FILTER_SETPASSALL(struct icmp6_filter*)

Pass all ICMPv6 messages to the application.

ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter*)

Block all ICMPv6 messages from the application.

ICMP6_FILTER_SETPASS(int, struct icmp6_filter*)

Pass messages of a certain ICMPv6 type to the application.

ICMP6_FILTER_SETBLOCK(int, struct icmp6_filter*)

Block messages of a certain ICMPv6 type from the application.

ICMP6_FILTER_WILLPASS(int, const struct icmp6_filter*)

Return true or false, depending on whether or not the specified message type is passed to the application.

ICMP6_FILTER_WILLBLOCK(int, const struct icmp6_filter*)

Return true or false, depending on whether or not the specified message type is blocked from the application.

When you create an ICMP6 raw socket, it passes all ICMPv6 message types to the application by default.

Based on:

RFC 2292

if_freenameindex()

Free dynamic memory allocated by if_nameindex()

Synopsis:

```
#include <net/if.h>

void if_freenameindex( struct if_nameindex * ptr );
```

Arguments:

ptr

A pointer to the if_nameindex structure to be freed.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The if_freenameindex() function frees the dynamic memory that you allocated by calling [if_nameindex\(\)](#) (p. 1052).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

if_indextoname()

Map an interface index to its name

Synopsis:

```
#include <net/if.h>

char * if_indextoname( unsigned int ifindex,
                      char * ifname );
```

Arguments:

- ifindex*
The interface index.
- ifname*
A pointer to a buffer in which *if_indextoname()* copies the interface name. The buffer must be a minimum of IFNAMSIZ bytes long.

Library:

libsocket
Use the -l socket option to gcc to link against this library.

Description:

The *if_indextoname()* function maps the interface index specified by *ifindex* to its corresponding name. The name is copied into the buffer pointed to by *ifname*.

Returns:

A pointer to the name, or NULL if There isn't an interface corresponding to the specified index.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

if_nameindex()

Return a list of interfaces

Synopsis:

```
#include <net/if.h>

struct if_nameindex * if_nameindex( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *if_nameindex()* function returns an array of *if_nameindex* structures, with one structure per interface, as defined in the include file *<net/if.h>*. The *if_nameindex* structure contains at least the following members:

unsigned int *if_index*

The index of the interface (1, 2, ...).

char **if_name*

A null-terminated name (e.g. *le0*).

The end of the array of structures is indicated by an entry with an *if_index* of 0 and an *if_name* of NULL.

Returns:

A valid array of *if_nameindex* structures, or NULL if an error occurred while using [getifaddrs\(\)](#) (p. 885) to retrieve the list, or there wasn't enough memory available.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

if_nametoindex()

Map an interface name to its index

Synopsis:

```
#include <net/if.h>

unsigned int if_nametoindex( const char * ifname );
```

Arguments:

ifname

The interface name that you want to map.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *if_nametoindex()* function maps the interface name specified by *ifname* to its corresponding index.

Returns:

The index number of the interface, or 0 if the specified interface couldn't be found or an error occurred while using [getifaddrs\(\)](#) (p. 885) to retrieve the list of interfaces.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ifaddrs

Structure that describes an Internet host

Synopsis:

```
#include <ifaddrs.h>

struct ifaddrs {
    struct ifaddrs *   ifa_next;
    char *             ifa_name;
    u_int              ifa_flags;
    struct sockaddr *   ifa_addr;
    struct sockaddr *   ifa_netmask;
    struct sockaddr *   ifa_dstaddr;
    void *             ifa_data;
};
```

Description:

The `ifaddrs` structure contains the following entries:

ifa_next

A pointer to the next structure in the list. This field is `NULL` in the last structure in the list.

ifa_name

The interface name.

ifa_flags

The interface flags, as set by the `ifconfig` utility.

ifa_addr

Either the address of the interface or the link-level address of the interface, if one exists; otherwise it's `NULL`. See the *sa_family* member of the `sockaddr` structure pointed to by *ifa_addr* to determine the format of the address.

ifa_netmask

The netmask associated with *ifa_addr*, if one is set; otherwise it's `NULL`.

ifa_dstaddr

The destination address on a P2P interface, if one exists; otherwise it's `NULL`. If the interface isn't a P2P interface, *ifa_dstaddr* contains the

broadcast address associated with *ifa_addr*, if one exists; otherwise it's NULL (see `<ifaddr.h>`).

ifa_data

Currently, this is set to NULL.

Classification:

[Unix](#) (p. 46)

ilogb(), ilogbf()

Compute the integral part of a logarithm

Synopsis:

```
#include <math.h>

int ilogb ( double x );
int ilogbf (float x );
```

Arguments:

x

The number you want to compute the integral part of the logarithm.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *ilogb()* and *ilogbf()* functions compute the integral part of:

$\log_r |x|$

as a signed integral value, for nonzero finite *x*, where *r* is the radix of the machine's floating point arithmetic.

Returns:

The exponent part of *x*, in integer format:

If <i>x</i> is:	<i>ilogb()</i> returns:
0	-INT_MAX
NAN	INT_MAX
Negative infinity	INT_MAX
Positive infinity	INT_MAX



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", ilogb(.5) );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

in8()

Read an 8-bit value from a port

Synopsis:

```
#include <hw/inout.h>
uint8_t in8( uintptr_t port );
```

Arguments:

port

The port you want to read the value from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *in8()* function reads an 8-bit value from the specified *port*.

Returns:

An 8-bit value.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [proc_mgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

in8s()

Read 8-bit values from a port

Synopsis:

```
#include <hw/inout.h>

void * in8s( void * buff,
             unsigned len,
             uintptr_t port );
```

Arguments:

buff

A pointer to a buffer where the function can store the values read.

len

The number of values that you want to read.

port

The port you want to read the values from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *in8s()* function reads *len* 8-bit values from the specified *port* and stores them in the buffer pointed to by *buff*.

Returns:

A pointer to the end of the read data.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:

Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

in16(), inbe16(), inle16()

Read a 16-bit value from a port

Synopsis:

```
#include <hw/inout.h>
uint16_t in16( uintptr_t port );
#define inbe16 ( port ) ...
#define inle16 ( port ) ...
```

Arguments:

port

The port you want to read the value from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *in16()* function reads a 16-bit value from the specified *port* in native-endian format (there's no conversion required).

The *inbe16()* and *inle16()* macros read a 16-bit value that's in big-endian or little-endian format, respectively, from the specified *port*, and returns the value as native-endian.



The *inbe16()* and *inle16()* macros access the specified port more than once if endian conversion is necessary. This could be a problem on some hardware.

Returns:

A 16-bit value in native-endian.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROCmgr_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NT0_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

Both `inbe16()` and `inle16()` are implemented as macros.

in16s()

Read 16-bit values from a port

Synopsis:

```
#include <hw/inout.h>

void * in16s( void * buff,
              unsigned len,
              uintptr_t port );
```

Arguments:

buff

A pointer to a buffer where the function can store the values read.

len

The number of values that you want to read.

port

The port you want to read the values from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *in16s()* function reads *len* 16-bit values from the specified *port* and stores them in the buffer pointed to by *buff*.

Returns:

A pointer to the end of the read data.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROCmgr_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

in32(), inbe32(), inle32()

Read a 32-bit value from a port

Synopsis:

```
#include <hw/inout.h>
uint32_t in32( uintptr_t port );
#define inbe32 ( port ) ...
#define inle32 ( port ) ...
```

Arguments:

port

The port you want to read the value from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *in32()* function reads a 32-bit value from the specified *port*.

The *inbe32()* and *inle32()* macros read a 32-bit value that's in big-endian or little-endian format, respectively, from the specified *port*, and returns the value as native-endian.



The *inbe32()* and *inle32()* macros access the specified port more than once if endian conversion is necessary. This could be a problem on some hardware.

Returns:

A 32-bit value in native-endian.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROCmgr_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

Both [inbe32\(\)](#) and [inle32\(\)](#) are implemented as macros.

in32s()

Read 32-bit values from a port

Synopsis:

```
#include <hw/inout.h>

void * in32s( void * buff,
              unsigned len,
              uintptr_t port );
```

Arguments:

buff

A pointer to a buffer where the function can store the values read.

len

The number of values that you want to read.

port

The port you want to read the values from.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *in32s()* function reads *len* 32-bit values from the specified *port* and stores them in the buffer pointed to by *buff*.

Returns:

A pointer to the end of the read data.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

index()

Find a character in a string

Synopsis:

```
#include <strings.h>

char* index( const char* s,
             int c );
```

Arguments:

s

The string you want to search. This string must end with a null (\0) character. The null character is considered to be part of the string.

c

The character you're looking for.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *index()* function returns a pointer to the first occurrence of the character *c* in the string *s*.

Returns:

A pointer to the character, or NULL if the character doesn't occur in the string.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

inet_addr()

Convert a string into a numeric Internet address

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr( const char * cp );
```

Arguments:

cp

A pointer to a string that represents an Internet address.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_addr()* routine converts a string representing an IPv4 Internet address (for example, “127.0.0.1”) into a numeric Internet address. To convert a hostname such as ftp.qnx.com, call [gethostbyname\(\)](#) (p. 872).

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

An Internet address, or INADDR_NONE if an error occurs.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

Although the value `INADDR_NONE` (0xFFFFFFFF) is a valid broadcast address, `inet_addr()` always indicates failure when returning that value. The [inet_aton\(\)](#) (p. 1075) function doesn't share this problem.

inet_aton()

Convert a string into an Internet address stored in a structure

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton( const char * cp,
               struct in_addr * addr );
```

Arguments:

cp

A pointer to the character string.

addr

A pointer to a `in_addr` structure where the function can store the converted address.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *inet_aton()* routine interprets the specified character string as an IPv4 Internet address, placing the address into the structure provided.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values.

For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

1

Success; the string was successfully interpreted.

0

Failure; the string is invalid.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet_lnaof()

Extract the local network address from an Internet address

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_lnaof( struct in_addr in );
```

Arguments:

in

An Internet address.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_lnaof()* routine returns the local network address for an IPv4 Internet address.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

A local network address.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet_makeaddr()

Convert a network number and a local network address into an Internet address

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr( unsigned long net,
                             unsigned long lna );
```

Arguments:

net

An Internet network number.

lna

The local network address.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *inet_makeaddr()* routine takes an Internet network number and a local network address and constructs an IPv4 Internet address.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

An Internet address.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet_net_ntop()

Convert an Internet network number to CIDR format

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char * inet_net_ntop( int af,
                     const void * src,
                     int bits,
                     char * dst,
                     size_t size );
```

Arguments:

af

The address family. Currently, only AF_INET is supported.

src

A pointer to the Internet network number that you want to convert. The format of the address is interpreted according to *af*.

bits

The number of bits that specify the network number (*src*).

dst

a pointer to the buffer where the function can store the converted address.

size

The size of the buffer that *dst* points to, in bytes.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_net_ntop()* function converts an Internet network number from network format (usually a struct `in_addr` or some other binary form, in network byte order) to

CIDR (Classless Internet Domain Routing) presentation format that's suitable for external display purposes.

With CIDR, a single IP address can be used to designate many unique IP addresses. A CIDR IP address looks like a normal IP address, except that it ends with a slash (/) followed by a number, called the *IP prefix*. For example:

172.200.0.0/16

The IP prefix specifies how many addresses are covered by the CIDR address, with lower numbers covering more addresses.

Network Numbers (IPv4 Internet addresses)

You can specify Internet addresses in the “dotted quad” notation, or Internet network numbers, using one of the following forms:

a.b.c.d/bits* or *a.b.c.d

When you specify a four-part address, each part is interpreted as a byte of data and is assigned, from left to right, to the four bytes of an Internet network number (or Internet address). When an Internet network number is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (i.e. right to left), such as the Intel 486 and Pentium processors, the bytes referred to above appear as “*d.c.b.a*”.

a.b.c

When you specify a three-part address, the last part is interpreted as a 16-bit quantity and is placed in the rightmost two bytes of the Internet network number (or network address). This makes the three-part address format convenient for specifying Class B network addresses as *net.net.host*.

a.b

When you specify a two-part address, the last part is interpreted as a 24-bit quantity and is placed in the rightmost three bytes of the Internet network number (or network address). This makes the two-part number format convenient for specifying Class A network numbers as *net.host*.

a

When you specify a one-part address, the value is stored directly in the Internet network number (network address) without any byte rearrangement.

All numbers supplied as “parts” in a dot notation may be decimal, octal, or hexadecimal, as specified in the C language. That is, a number is interpreted as decimal unless it has a leading 0 (octal), or a leading 0x or 0X (hex).

Returns:

A pointer to the destination string (*dst*), or NULL if a system error occurs (*errno* (p. 539) is set).

Errors:**ENOENT**

Invalid argument *af*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

inet_netof()

Extract the network number from an Internet address

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_netof( struct in_addr in );
```

Arguments:

in

An Internet address.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_netof()* routine returns the network number of the specified IPv4 Internet address.

All Internet addresses are returned in network order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

An Internet network number.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

inet_net_pton()

Convert an Internet network number from CIDR format to network format

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_net_pton( int af,
                  const char * src,
                  void * dst,
                  size_t size );
```

Arguments:

af

The address family. Currently, only AF_INET is supported.

src

A pointer to the presentation-format (CIDR) address. The format of the address is interpreted according to *af*.

dst

A pointer to the buffer where the function can store the converted address.

size

The size of the buffer pointed to by *dst*, in bytes.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_net_pton()* function converts an Internet network number from presentation format — a printable form as held in a character string, such as, Internet standard dot notation, or Classless Internet Domain Routing (CIDR) — to network format (usually a struct `in_addr` or some other internal binary representation, in network byte order).

For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1080).

Returns:

The number of bits that specify the network number (computed based on the class, or specified with /CIDR), or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**ENOENT**

Invalid argument *af*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

inet_network()

Convert a string into an Internet network number

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network( const char * cp );
```

Arguments:

cp

A pointer to a string representing an Internet address.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_network()* routine converts a string representing an IPv4 Internet address (for example, “127.0.0.1”) into a numeric Internet network number.

All Internet addresses are returned in network order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

An Internet network number, or INADDR_NONE if an error occurs.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

inet_ntoa()*Convert an Internet address into a string***Synopsis:**

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char * inet_ntoa( struct in_addr in );
```

Arguments:*in*

The Internet address that you want to convert.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_ntoa()* routine converts an IPv4 Internet address into an ASCII string representing the address in dot notation (for example, “127.0.0.1”).

For more information on Internet addresses, see [inet_net_ntop\(\)](#) (p. 1081).

Returns:

A string representing an Internet address.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The string returned by this function is stored in a static buffer that's reused for every call to *inet_ntoa()*. For a thread-safe version, see [inet_ntoa_r\(\)](#) (p. 1091).

inet_ntoa_r()

Convert an Internet address into a string

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char * inet_ntoa_r( struct in_addr in,
                   char * buffer,
                   int buflen );
```

Arguments:

in

The Internet address that you want to convert.

buffer

A buffer where the function can store the result.

buflen

The size of the buffer, in bytes.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_ntoa_r()* function is a thread-safe version of [inet_ntoa\(\)](#) (p. 1089). It converts an IPv4 Internet address into a string (for example, "127.0.0.1"). For more information on this routine, see [inet_aton\(\)](#) (p. 1075).

Returns:

A string representing an Internet address, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

ERANGE

The supplied *buffer* isn't large enough to store the result.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

inet_ntop()

Convert a numeric network address to a string

Synopsis:

```
#include <sys/socket.h>
#include <arpa/inet.h>

const char * inet_ntop( int af,
                        const void * src,
                        char * dst,
                        socklen_t size );
```

Arguments:

af

The *src* address's network family; one of:

AF_INET

IPv4 addresses

AF_INET6

IPv6 addresses

src

The numeric network address that you want to convert to a string.

dst

The text string that represents the translated network address. You can use the following constants to allocate buffers of the correct size (they're defined in <netinet/in.h>):

- INET_ADDRSTRLEN — storage for an IPv4 address
- INET6_ADDRSTRLEN — storage for an IPv6 address

size

The size of the buffer pointed to by *dst*.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `inet_ntop()` function converts a numeric network address pointed to by `src` into a text string in the buffer pointed to by `dst`.

Returns:

A pointer to the buffer containing the text version of the address, or `NULL` if an error occurs (`errno` (p. 539) is set).

Errors:

EINVAL

The value of the `af` argument isn't a supported network family.

ENOSPC

The `dst` buffer isn't large enough (according to `size`) to store the translated address.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>

#define INADDR "10.1.0.29"
#define IN6ADDR "DEAD:BEEF:7654:3210:FEDC:3210:7654:BA98"

int
main()
{
    struct in_addr inaddr;
    struct in6_addr in6addr;
    char buf[INET_ADDRSTRLEN], buf6[INET6_ADDRSTRLEN];
    int rval;

    if ( (rval = inet_pton(AF_INET, INADDR, &inaddr)) == 0 ) {
        printf("Invalid address: %s\n", INADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(AF_INET, &inaddr, buf, sizeof(buf)) != NULL)
        printf("inet addr: %s\n", buf);
    else {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }

    if ( (rval = inet_pton(AF_INET6, IN6ADDR, &in6addr)) == 0 ) {
        printf("Invalid address: %s\n", IN6ADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }
}
```

```
    }  
    if (inet_ntop(AF_INET6, &in6addr, buf6, sizeof(buf6)) != NULL)  
        printf("inet6 addr: %s\n", buf6);  
    else {  
        perror("inet_ntop");  
        exit(EXIT_FAILURE);  
    }  
    return(EXIT_SUCCESS);  
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet_pton()

Convert a text host address to a numeric address

Synopsis:

```
#include <sys/socket.h>
#include <arpa/inet.h>

int inet_pton( int af,
               const char * src,
               void * dst );
```

Arguments:

af

The *src* address's network family; one of:

AF_INET

IPv4 addresses

AF_INET6

IPv6 addresses

src

A pointer to the text host address that you want to convert. The format of the address is interpreted according to *af*

dst

A pointer to a buffer where the function can store the converted address.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *inet_pton()* function converts the standard text representation of the numeric network address (*src*) into its numeric network byte-order binary form (*dst*).

The converted address is stored in network byte order in *dst*. The buffer pointed to by *dst* must be large enough to hold the numeric address:

Family	Numeric address size
AF_INET	4 bytes
AF_INET6	16 bytes

AF_INET addresses

IPv4 addresses must be specified in the standard dotted-decimal form:

ddd.ddd.ddd.ddd

where *ddd* is a one- to three-digit decimal number between 0 and 255.



Many existing implementations of [inet_addr\(\)](#) (p. 1073) and [inet_aton\(\)](#) (p. 1075) accept nonstandard input: octal numbers, hexadecimal numbers, and fewer than four numbers. The [inet_pton\(\)](#) function doesn't accept these formats.

AF_INET6 addresses

IPv6 addresses must be specified in one of the following standard formats:

- The preferred form is:

x:x:x:x:x:x:x

where *x* is a hexadecimal value for one of the eight 16-bit pieces of the address.

For example:

- DEAD:BEEF:7654:3210:FEDC:3210:7654:BA98
- 417A:200C:800:8:0:0:0:1080
- A :: can be used once per address to represent multiple groups of 16 zero-bits. For example, the following addresses:

- 1080:0:0:0:8:800:200C:417A
- FF01:0:0:0:0:0:0:43
- 0:0:0:0:0:0:0:1
- 0:0:0:0:0:0:0:0

can be represented as:

- 1080::8:800:200C:417A
- FF01::43
- ::1
- ::
- A convenient format when dealing with mixed IPv4 and IPv6 environments is:

x:x:x:x:x:d.d.d.d

where *x* is a hexadecimal value for one of the six high-order 16-bit pieces of the address and *d* is a decimal value for one of the four low-order 8-bit pieces of the address (standard [AF_INET](#) (p. 1097) representation). For example:

- 0:0:0:0:0:0:13.1.68.3
- 0:0:0:0:0:FFFF:129.144.52.38

Or, in their compressed forms:

- ::13.1.68.3
- ::FFFF:129.144.52.38

Based on:

RFC 2373

Returns:

1

Success.

0

The input isn't a valid address.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAFNOSUPPORT

The *af* argument isn't one of the supported networking families.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>

#define INADDR "10.1.0.29"
#define IN6ADDR "DEAD:BEEF:7654:3210:FEDC:3210:7654:BA98"

int
main()
{
    struct in_addr inaddr;
    struct in6_addr in6addr;
    char buf[INET_ADDRSTRLEN], buf6[INET6_ADDRSTRLEN];
    int rval;

    if ( (rval = inet_pton(AF_INET, INADDR, &inaddr)) == 0 ) {
```

```

        printf("Invalid address: %s\n", INADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(AF_INET, &inaddr, buf, sizeof(buf)) != NULL)
        printf("inet addr: %s\n", buf);
    else {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }

    if ( (rval = inet_pton(AF_INET6, IN6ADDR, &in6addr)) == 0) {
        printf("Invalid address: %s\n", IN6ADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(AF_INET6, &in6addr, buf6, sizeof(buf6)) != NULL)
        printf("inet6 addr: %s\n", buf6);
    else {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }

    return(EXIT_SUCCESS);
}

```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

INET6

Internet protocol version 6 family

Synopsis:

```
#include <netinet/in.h>

struct sockaddr_in6 {
    uint8_t sin6_len;
    sa_family_t sin6_family;
    in_port_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
};
```

Description:

Protocols

The INET6 family consists of the:

- IPv6 network protocol
- Internet Control Message Protocol version 6 ([ICMP](#) (p. 1045))
- Transmission Control Protocol ([TCP](#) (p. 3359))
- User Datagram Protocol ([UDP](#) (p. 3553)).

TCP supports the `SOCK_STREAM` abstraction, while UDP supports the `SOCK_DGRAM` abstraction. Note that TCP and UDP are common to INET and INET6. A raw interface to IPv6 is available by creating an Internet `SOCK_RAW` socket. The ICMPv6 message protocol may be accessed from a raw socket.

The INET6 protocol family is an updated version of the INET family. While INET implements Internet Protocol version 4, INET6 implements Internet Protocol version 6.

Addressing

IPv6 addresses are 16-byte quantities, stored in network standard (big-endian) byte order. The header file `<netinet/in.h>` defines this address as a discriminated union.

Sockets bound to the INET6 family use the structure shown above.

You can create sockets with the local address `::` (which is equal to IPv6 address `0:0:0:0:0:0:0:0`) to cause “wildcard” matching on incoming messages. You can specify the address in a call to [connect\(\)](#) (p. 348) or [sendto\(\)](#) (p. 2837) as `::` to mean the local host. You can get the `::` value by setting the `sin6_addr` field to 0, or by using the

address contained in the *in6addr_any* global variable, which is declared in `<netinet6/in6.h>`.

The IPv6 specification defines *scoped addresses*, such as link-local or site-local addresses. A scoped address is ambiguous to the kernel if it's specified without a scope identifier. To manipulate scoped addresses properly in your application, use the advanced API defined in *RFC 2292*. A compact description on the advanced API is available in *IP6* (p. 1412). If you specify scoped addresses without an explicit scope, the socket manager may return an error.



Scoped addresses are currently experimental, from both a specification and an implementation point of view.

The KAME implementation supports extended numeric IPv6 address notation for link-local addresses. For example, you can use `fe80::1%de0` to specify “fe80::1 on the de0 interface.” The *getaddrinfo()* (p. 818) and *getnameinfo()* functions support this notation. Some utilities, such as `telnet` and `ftp`, can use the notation. With special programs like `ping6`, you can disambiguate scoped addresses by specifying the outgoing interface with extra command-line options.

The socket manager handles scoped addresses in a special manner. In the socket manager's routing tables or interface structures, a scoped address's interface index is embedded in the address. Therefore, the address contained in some of the socket manager structures isn't the same as on the wire. The embedded index becomes visible when using the `PF_ROUTE` socket or the *sysctl()* (p. 3311) function. You shouldn't use the embedded form.

Interaction between IPv4/v6 sockets

The behavior of the `AF_INET6` TCP/UDP socket is documented in the *RFC 2553* specification, which states:

- A specific bind on an `AF_INET6` socket (*bind()* (p. 208) with an address specified) should accept IPv6 traffic to that address only.
- If you perform a wildcard bind on an `AF_INET6` socket (*bind()* (p. 208) to the IPv6 address `::`), and there isn't a wildcard-bound `AF_INET` socket on that TCP/UDP port, then the IPv6 traffic as well as the IPv4 traffic should be routed to that `AF_INET6` socket. IPv4 traffic should be seen by the application as if it came from an IPv6 address such as `::ffff:10.1.1.1`. This is called an *IPv4 mapped address*.
- If there are both wildcard-bound `AF_INET` sockets and wildcard-bound `AF_INET6` sockets on one TCP/UDP port, they should operate independently: IPv4 traffic should be routed to the `AF_INET` socket, and IPv6 should be routed to the `AF_INET6` socket.

However, the *RFC 2553* specification doesn't define the constraint between the binding order, nor how the IPv4 TCP/UDP port numbers and the IPv6 TCP/UDP port numbers relate each other (whether they must be integrated or separated). The behavior is very

different from implementation to implementation. It is unwise to rely too much on the behavior of the AF_INET6 wildcard-bound socket. Instead, connect to two sockets, one for AF_INET and another for AF_INET6, when you want to accept both IPv4 and IPv6 traffic.



Use caution when handling connections from IPv4 mapped addresses with AF_INET6 sockets—if the target node routes IPv4 traffic to AF_INET6 sockets, malicious parties can bypass security.

Because of the security hole, by default, NetBSD doesn't route IPv4 traffic to AF_INET6 sockets. If you want to accept both IPv4 and IPv6 traffic, use two sockets. IPv4 traffic may be routed with multiple per-socket/per-node configurations, but, it isn't recommended. See [IP6](#) (p. 1412) for details.



The IPv6 support is subject to change as the Internet protocols develop. Don't depend on details of the current implementation, but rather the services exported. Try to implement version-independent code as much as possible, because you'll need to support both INET and INET6.

Based on:

RFC 2553, RFC 2292

inet6_option_alloc()

Append IPv6 hop-by-hop or destination options into ancillary data object

Synopsis:

```
#include <netinet/in.h>

u_int8_t * inet6_option_alloc(struct cmsghdr *cmsg,
                             int datalen,
                             int multx,
                             int plusy);
```

Arguments:

cmsg

A pointer to the cmsghdr structure that must have been initialized by *inet6_option_init()*.

datalen

The length of the option, in bytes. This value is required as an argument to allow the function to determine if padding should be appended at the end of the option, argument since the option data length must already be stored by the caller (the *inet6_option_append()* function doesn't need a data length).

multx

The value x in the alignment term $xn + y$. It must have a value of 1, 2, 4, or 8.

plusy

Value y in the alignment term $xn + y$. It must have a value between 0 and 7, inclusive.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This *inet6_option_alloc()* function appends a hop-by-hop option or a destination option into an ancillary data object that has been initialized by *inet6_option_init()*.

The difference between this function and *inet6_option_append()* is that the latter copies the contents of the previously built option into the ancillary data object. This function returns a pointer to the space in the data object where the option's type-length-value or TLV must then be built by the caller.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

A pointer to the 8-bit option type field that starts the option, or NULL if an error has occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_option_append()

Append an IPv6 hop-by-hop or destination option to an ancillary data object

Synopsis:

```
#include <netinet/in.h>

int inet6_option_append(struct cmsghdr *cmsg,
                       const u_int8_t *typep,
                       int multx,
                       int plusy);
```

Arguments:

cmsg

A pointer to the cmsghdr structure that must have been initialized by *inet6_option_init()*.

typep

A pointer to the 8-bit option type. It's assumed that this field is immediately followed by the 8-bit option data length field, which is then followed by the option data. You must initialize these three fields (the type-length-value, or TLV) before calling this function.

The option type must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the Pad1 and PadN options, respectively.)

The option data length must be between 0 and 255, inclusive, and is the length of the option data that follows.

multx

The value *x* in the alignment term $xn + y$. It must have a value of 1, 2, 4, or 8.

plusy

The value *y* in the alignment term $xn + y$. It must have a value between 0 and 7, inclusive.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

This *inet6_option_append()* function appends a hop-by-hop option or a destination option to an ancillary data object that has been initialized by *inet6_option_init()*.

See also:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

0

Success.

-1

An error has occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_option_find()

Search for IPv6 hop-by-hop and destination options

Synopsis:

```
#include <netinet/in.h>

int inet6_option_find(const struct cmsghdr *cmsg,
                     u_int8_t **tptrp,
                     int type);
```

Arguments:

cmsg

A pointer to the `cmsghdr` structure that must have been initialized by `inet6_option_init()`.

type

The type of option to search for. Either `IPV6_HOPOPTS` or `IPV6_DSTOPTS`. This type is stored in the `cmsg_type` member of the `cmsghdr` structure pointed to by `*cmsgp`.

tptrp

A pointer to a pointer to an 8-bit byte.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

This `inet6_option_find()` function is similar to `inet6_option_next()`. It however, lets the caller specify the option type to be searched for, instead of always returning the next option in the ancillary data object. The `cmsg` is a pointer to the `cmsghdr` structure of which `cmsg_level` equals `IPPROTO_IPV6` and `cmsg_type` equals either `IPV6_HOPOPTS` or `IPV6_DSTOPTS`.

The `tptrp` is a pointer to a pointer to an 8-bit byte that the function uses to remember its place in the ancillary data object each time the function is called.

The first time you call this function for a given ancillary data object, you must set **tprp* must be set to NULL. This function starts searching for an option of the specified type beginning after the value of **tprp* pointer.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

0 with **tprp* pointing to the 8-bit option

The option was found.

-1 with **tprp* pointing to NULL

The option wasn't found.

-1 with **tprp* pointing to non-NULL

An error has occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_option_init()

Initialize an ancillary data object that contains IPv6 hop-by-hop and destination options

Synopsis:

```
#include <netinet/in.h>

int inet6_option_init(void *bp,
                     struct cmsghdr **cmsgp,
                     int type);
```

Arguments:

bp

A pointer to previously allocated space that contains the ancillary data object. It must be large enough to contain all the individual options to be added by later calls to *inet6_option_append()* and *inet6_option_alloc()*.

cmsgp

A pointer to a cmsghdr structure. The **cmsgp* variable is initialized by this function to point to the cmsghdr structure that this function constructs in the buffer pointed to by *bp*.

type

The type of option which must be either IPV6_HOPOPTS or IPV6_DSTOPTS. This type is stored in the *cmsg_type* member of the cmsghdr structure pointed to by **cmsgp*.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

Call *inet6_option_init()* function once per ancillary data object that contains either hop-by-hop or destination options.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains examples.

- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:**0**

Success.

-1

An error has occurred.

Classification:[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_option_next()

Find the next IPv6 hop-by-hop or destination option

Synopsis:

```
#include <netinet/in.h>

int inet6_option_next(const struct cmsghdr *cmsg,
                     u_int8_t **tptrp);
```

Arguments:

cmsg

A pointer to the cmsghdr structure that must have been initialized by *inet6_option_init()*.

tptrp

A pointer to a pointer to an 8-bit byte.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This *inet6_option_next()* function finds the next hop-by-hop option or destination option in an ancillary data object. If another option remains to be processed, the return value of the function is 0 and **tptrp* points to the 8-bit option type field the option data.

The *cmsg* variable is a pointer to cmsghdr structure for which *cmsg_level* equals IPPROTO_IPV6 and *cmsg_type* equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

The *tptrp* is a pointer to a pointer to an 8-bit byte and **tptrp* is used by the function to remember its place in the ancillary data object each time the function is called. The first time you call this function for a given ancillary data object, you must set **tptrp* to NULL.

Each time this function returns success, **tptrp* points to the 8-bit option type field for the next option to be processed.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

0

The option is located and the **tptrp* points to the 8-bit option type field.

-1 with **tptrp* pointing to NULL

No more options to process.

-1 with **tptrp* pointing to non-NULL

An error has occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_option_space()

Determine how much space an IPv6 hop-by-hop or destination option requires

Synopsis:

```
#include <netinet/in.h>

int inet6_option_space(int nbytes);
```

Arguments:

nbytes

The size of the structure that defines the option. It includes any padding bytes at the beginning (the value y in the alignment term $xn + y$, the type byte), the length byte, and the option data.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

This *inet6_option_space()* function returns the number of bytes required to hold an option when it's stored as ancillary data, including the `cmsghdr` structure at the beginning, and any padding at the end (to make its size a multiple of 8 bytes).



When multiple options are stored in a single ancillary data object, this function overestimates the amount of space required by the size of $N-1$ `cmsghdr` structures, where N is the number of options to be stored in the object. This is of little consequence, since it's assumed that most hop-by-hop option and destination option headers carry only one option (see Appendix B of *RFC 2460*).

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_add()

Add an address to an IPv6 routing header

Synopsis:

```
#include <netinet/in.h>

int inet6_rthdr_add(struct cmsghdr *cmsg,
                   const struct in6_addr *addr,
                   unsigned int flags);
```

Arguments:

addr

A pointer to the IPv6 address structure to add to the routing header.

flags

Routing header flags. For an IPv6 Type 0 routing header, it's either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

cmsg

A pointer to Ancillary data containing the routing header.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This function adds the address pointed to by *addr* to the end of the Routing header being constructed and sets the type of this hop to the value of *flags*.

If successful, the *cmsg_len* member of the *cmsg* structure is updated to account for the new address in the routing header.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:**0**

Success.

-1

An error has occurred.

Classification:[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_getaddr()

Get a pointer to an IPv6 address in the routing header

Synopsis:

```
#include <netinet/in.h>

struct in6_addr * inet6_rthdr_getaddr(
    struct cmsghdr *cmsg,
    int index);
```

Arguments:

cmsg

A pointer to the Ancillary data containing the routing header.

index

A value between 0 and the number returned by *inet6_rthdr_segments()*.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This function returns a pointer to the IPv6 address specified by *index* in the routing header described by *cmsg*. The *index* must have a value between 1 and the number returned by *inet6_rthdr_segments()*. You should first call *inet6_rthdr_segments()* to obtain the number of segments in the Routing header.

For more information, see:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

A pointer to the IPv6 address, or NULL if an error occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_getflags()

Get the flags for a segment in an IPv6 routing header

Synopsis:

```
#include <netinet/in.h>

int inet6_rthdr_getflags(const struct cmsghdr *cmsg,
                        int index);
```

Arguments:

cmsg

A pointer to the Ancillary data containing the routing header.

index

A value between 0 and the number returned by *inet6_rthdr_segments()*.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This function returns the flags for the segment specified by *index* in the routing header described by *cmsg*. The *index* must have a value between 0 and the number returned by *inet6_rthdr_segments()*.



Addresses are indexed starting at 1, and flags starting at 0. They're consistent with the terminology and figures in *RFC2460*.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

- IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT for an IPv6 Type 0 routing header

- -1 on error.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_init()

Initialize an IPv6 routing header

Synopsis:

```
#include <netinet/in.h>

struct cmsghdr * inet6_rthdr_init(void *bp,
                                  int type);
```

Arguments:

bp

A pointer to the buffer where the function can build a `cmsghdr` structure followed by a Routing header of the specified type.

type

The type of IPv6 Routing header (e.g. Type 0 as defined in `<netinet/in.h>`).

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

This function initializes the buffer pointed to by *bp* to contain a `cmsghdr` structure followed by a Routing header of the specified type. The `cmsg_len` member of the `cmsghdr` structure is initialized to the size of the structure plus the amount of space required by the Routing header.

The `cmsg_level` and `cmsg_type` members are also initialized as required.

You must allocate the buffer before calling this function. To determine the size of the buffer, call `inet6_rthdr_space()`.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

A pointer to the `cmsghdr` structure, which you'll pass to other functions (and used as the first argument to list functions) or NULL if an error occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_lasthop()

Specify the Strict/Loose flag for the final hop of an IPv6 routing header

Synopsis:

```
#include <netinet/in.h>

int inet6_rthdr_lasthop(struct cmsghdr *cmsg,
                       unsigned int flags);
```

Arguments:

cmsg

Ancillary data containing routing header.

flags

Routing header flags. It's either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT for an IPv6 Type 0 routing header.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This function specifies the Strict/Loose flag for the final hop of a Routing header.



A routing header specifying N intermediate nodes requires $N+1$ Strict/Loose flags. This requires N calls to *inet6_rthdr_add()* followed by one call to *inet6_rthdr_lasthop()*.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

0

Success.

-1

An error has occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_reverse()

Reverse the list of addresses in an IPv6 router header

Synopsis:

```
#include <netinet/in.h>

int inet6_rthdr_reverse(const struct cmsghdr *in,
                       struct cmsghdr *out);
```

Arguments:

in

Ancillary data containing Routing header.

out

Ancillary data containing Routing header.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:



The `inet6_rthdr_reverse()` has not been implemented yet.

This function takes a routing header that has been received as ancillary data (pointed to by the first argument, *in*) and writes a new routing header. The routing header sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

0

Success.

-1

An error has occurred.

Classification:[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_segments()

Count the segments in an IPv6 routing header

Synopsis:

```
#include <netinet/in.h>

int inet6_rthdr_segments(const struct cmsghdr *cmsg);
```

Arguments:

cmsg

A pointer to Ancillary data containing a routing header.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

inet6_rthdr_segments()

This function returns the number of segments (addresses) contained in the Routing header described by *cmsg*.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.
- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

1 to 23

Success.

-1

An error has occurred.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

inet6_rthdr_space()

Determine the space required by an IPv6 routing header

Synopsis:

```
#include <netinet/in.h>

size_t inet6_rthdr_space(int type,
                        int segments);
```

Arguments:

type

The type of IPv6 Routing header (e.g. Type 0 as defined in <netinet/in.h>).

segments

The number of segments (addresses) in the Routing header.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

This function returns the number of bytes required to hold a Routing header of the specified type containing a specified number of segments (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 1 and 23, inclusive. The return value includes the size of the cmsghdr structure that precedes the Routing header, and any required padding.



This function returns the size but doesn't allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired. All the ancillary data objects must be specified to *sendmsg()* as a single *msg_control* buffer in the *msg_hdr* structure *msg_control* member.

Based on:

- W. Stevens and M. Thomas, Advanced Sockets API for IPv6, *RFC 2292*, February 1998. Contains good examples.

- S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, *RFC 2460*, December 1998.

Returns:

0, for either of the two situations: the type of the routing header isn't supported by this implementation or the number of segments is invalid for this type of routing header.

Classification:

[RFC 2292](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

initgroups()

Initialize the supplementary group access list

Synopsis:

```
#include <grp.h>
#include <sys/types.h>

int initgroups( const char * name,
                gid_t basegid );
```

Arguments:

name

The name of the user whose group membership you want to use as the supplementary group access list.

basegid

A group ID that you want to include in the group access list.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *initgroups()* function reads the group membership for the user specified by *name* from the group database, and then initializes the supplementary group access list of the calling process (see [getgrnam\(\)](#) (p. 857) and [getgroups\(\)](#) (p. 864)).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EPERM

The caller isn't root.

Files:

/etc/group

The group database.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

If *initgroups()* fails, it *doesn't* change the supplementary group access list.

The *getgrouplist()* function called by *initgroups()* is based on [getgrent\(\)](#) (p. 851). If the calling process uses *getgrent()*, the in-memory group structure is overwritten in the call to *initgroups()*.

initstate()

Initialize a pseudo-random number generator

Synopsis:

```
#include <stdlib.h>

char* initstate( unsigned int seed,
                 char* state,
                 size_t size );
```

Arguments:

seed

A starting point for the random-number sequence. This lets you restart the sequence at the same point.

state

The state array that you want to initialize.

size

The size, in bytes, of the state array; see below.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *initstate()* initializes the given state array for future use when generating pseudo-random numbers.

This function uses the *size* argument to determine what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. For values smaller than 8, *random()* uses a simple linear congruential random number generator.

Use this function in conjunction with the following:

***random()* (p. 2487)**

Generate a pseudo-random number using a default state.

***setstate()* (p. 2913)**

Specify the state of the pseudo-random number generator.

***srandom()* (p. 3158)**

Set the seed used by the pseudo-random number generator.

If you haven't called *initstate()*, *random()* behaves as though you had called *initstate()* with a *seed* of 1 and a *size* of 128.

After initialization, you can restart a state array at a different point in one of these ways:

- Call *initstate()* with the desired seed, state array, and size of the array.
- Call *setstate()* with the desired state, then call *srandom()* with the desired seed. The advantage of using both of these functions is that the size of the state array doesn't have to be saved once it's initialized.

Returns:

A pointer to the previous state array, or NULL if an error occurred.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

static char state1[32];

int main() {
    initstate( time(NULL), state1, sizeof(state1));
    setstate(state1);
    printf("%d0\n", random());
    return EXIT_SUCCESS;
}
```

Classification:

***POSIX 1003.1 XSI* (p. 46)**

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

inotify_add_watch()

Add or update a watch for filesystem events associated with a path

Synopsis:

```
#include <sys/inotify.h>

int inotify_add_watch( int fd,
                     const char *pathname,
                     int mask );
```

Arguments:

fd

A valid file descriptor returned by [inotify_init\(\)](#) (p. 1139).

path

The path whose filesystem events you want to monitor.

mask

A bitmask that specifies the events you want to watch. The bits include:

- `IN_ACCESS` — the file was read.
- `IN_MODIFY` — the file was written to.
- `IN_ATTRIB` — the attributes of the file changed.
- `IN_CLOSE_WRITE` — a file that was opened for writing was closed.
- `IN_CLOSE_NOWRITE` — a file that was opened not for writing was closed.
- `IN_CLOSE` — the same as `(IN_CLOSE_WRITE | IN_CLOSE_NOWRITE)`.
- `IN_OPEN` — the file was opened.
- `IN_MOVED_FROM` — the file was moved or renamed away from the item being watched.
- `IN_MOVED_TO` — the file was moved or renamed to the item being watched.
- `IN_MOVE` — the same as `(IN_MOVED_FROM | IN_MOVED_TO)`.
- `IN_CREATE` — a file was created in a watched directory.
- `IN_DELETE` — a file or directory was deleted.
- `IN_DELETE_SELF` — the file or directory being monitored was deleted.
- `IN_MOVE_SELF` — the file or directory being monitored was moved or renamed.

IN_ALL_EVENTS is a bitwise-OR of all the event types.

You can OR the following into the event type:

- IN_ONESHOT — remove the watch during the generation of the first event.
- IN_ONLYDIR — watch the pathname only if it's a directory.
- IN_DONT_FOLLOW — don't dereference the pathname if it's a symbolic link.
- IN_EXCL_UNLINK — don't generate events for children after they've been unlinked from the watched directory.
- IN_MASK_ADD — add events to the watch mask (if one already exists) for this pathname, instead of replacing it.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `inotify_add_watch()` function starts watching for filesystem events associated with the given path.

For an overview of inotify, see <http://www.linuxjournal.com/article/8478?page=0,0> in the Linux Journal, but note that there are differences between the Linux and QNX Neutrino implementations.

Returns:

A watch descriptor, or -1 if an error occurred ([errno](#) (p. 539) is set).

Classification:

[Linux](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

inotify_event

Structure that describes a watched filesystem event

Synopsis:

```
#include <sys/inotify.h>

struct inotify_event {
    _Int32t    wd;
    _Uint32t   mask;
    _Uint32t   cookie;
    _Uint32t   len;
    char name[0];
};
```

Description:

The `inotify_event` structure describes a filesystem event returned by the `inotify` system. You get these events by reading the file descriptor returned by [inotify_init\(\)](#) (p. 1139).

The members of this structure include:

wd

The watch descriptor associated with the event, as returned by [inotify_add_watch\(\)](#) (p. 1135).

mask

A bitmask that includes the event type:

- `IN_ACCESS` — the file was read.
- `IN_MODIFY` — the file was written to.
- `IN_ATTRIB` — the attributes of the file changed.
- `IN_CLOSE_WRITE` — a file that was opened for writing was closed.
- `IN_CLOSE_NOWRITE` — a file that was opened not for writing was closed.
- `IN_OPEN` — the file was opened.
- `IN_MOVED_FROM` — the file was moved or renamed away from the item being watched.
- `IN_MOVED_TO` — the file was moved or renamed to the item being watched.
- `IN_CREATE` — a file was created in a watched directory.
- `IN_DELETE` — a file or directory was deleted.
- `IN_DELETE_SELF` — the file or directory being monitored was deleted.

- `IN_MOVE_SELF` — the file or directory being monitored was moved or renamed.

along with other information:

- `IN_UNMOUNT` — the backing filesystem was unmounted.
- `IN_Q_OVERFLOW` — the inotify queue overflowed.
- `IN_IGNORED` — the watch was automatically removed because the file was deleted, or its filesystem was unmounted.
- `IN_ISDIR` — the event occurred against a directory.

cookie

An unique number that identifies related events.

For example, if you're watching two directories, and a file is moved from one of the directories into the other, you get two separate events: an `IN_MOVED_FROM` for one watched directory, and an `IN_MOVED_TO` for the other. Both events have the same *cookie* value, to indicate that they're related.

len

The length of the *name* field, including any required padding.

name

The name of the object that the event occurred to.

While the QNX Neutrino inotify implementation follows the same semantics as the Linux implementation, the behavior differs from Linux due to the kernel and filesystem models employed. For instance, you may see `IN_OPEN`, `IN_STAT`, and `IN_CLOSE` events for a single invocation of `stat()`. For an overview of inotify, see <http://www.linuxjournal.com/article/8478?page=0,0> in the Linux Journal.

Classification:

[Linux](#) (p. 46)

inotify_init()

Initialize the inode notify system

Synopsis:

```
#include <sys/inotify.h>

int inotify_init( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *inotify_init()* function initializes the inode notify system that you can use to watch for filesystem events.



This function opens the file `/dev/fsnotify`. If your process doesn't have read permission for this path, the function fails.

The basic steps for using the inotify system are as follows:

- Initialize the inotify system by calling *inotify_init()*, saving the file descriptor that it returns.
- Use *inotify_add_watch()* (p. 1135) to add or modify a watch for a given file or directory.
- Read the filesystem events (of type `struct inotify_event` (p. 1137)) from the file descriptor returned by *inotify_init()*.
- To remove a watch, call *inotify_rm_watch()* (p. 1141).
- To remove all watches and shut down inotify, close the file descriptor returned by *inotify_init()*.

While the QNX Neutrino inotify implementation follows the same semantics as the Linux implementation, the behavior differs from Linux due to the kernel and filesystem models employed. For instance, you may see `IN_OPEN`, `IN_STAT`, and `IN_CLOSE` events for a single invocation of *stat()*. For an overview of inotify, see <http://www.linuxjournal.com/article/8478?page=0,0> in the Linux Journal.

Returns:

A file descriptor for the inotify mailbox, or -1 if an error occurred (*errno* (p. 539) is set).

Classification:

Linux (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

inotify_rm_watch()

Remove the watch associated with the given watch descriptor

Synopsis:

```
#include <sys/inotify.h>

int inotify_rm_watch( int fd,
                    int32_t wd );
```

Arguments:

fd

A valid file descriptor returned by [inotify_init\(\)](#) (p. 1139).

wd

A valid watch description returned by [inotify_rm_watch\(\)](#) (p. 1141).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *inotify_rm_watch()* function removes a watch that you earlier set to watch for filesystem events.

For an overview of inotify, see <http://www.linuxjournal.com/article/8478?page=0,0> in the Linux Journal, but note that there are differences between the Linux and QNX Neutrino implementations.

Returns:

0 on success, or -1 if an error occurred ([errno](#) (p. 539) is set).

Classification:

[Linux](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

input_line()

Get a string of characters from a file

Synopsis:

```
#include <stdio.h>

char* input_line( FILE* fp,
                  char* buf,
                  int bufsize );

extern int _input_line_max;
```

Arguments:

fp

The file that you want to read from.

buf

A pointer to a buffer where the function can store the string that it reads.

bufsize

The size of the buffer, in bytes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *input_line()* function gets a string of characters from the file designated by *fp* and stores them in the array pointed to by *buf*. The *input_line()* function stops reading characters when:

- end-of-file is reached
- a newline character is read
- *bufsize* - 1 characters have been read.

In addition, the *input_line()* function buffers the last *_input_line_max* lines internally. The *_input_line_max* variable is defined in `<stdio.h>`. You can set it before calling *input_line()* for the first time; its default value is 20. While the line is being read, the **KEY_UP** and **KEY_DOWN** keys can be used to move to the previous and next line respectively in a circular buffer of previously read lines. The newline character (`\n`) is replaced with the null character on input.

Returns:

A pointer to the input line. On end-of-file or on encountering an error reading from *fp*, NULL is returned and *errno* (p. 539) is set.

Examples:

```
#include <stdlib.h>
#include <stdio.h>

#define SIZ 256

int _input_line_max;

int main( void )
{
    FILE    *fp;
    char    *p,
            buf[SIZ];

    fp = stdin;          /* Or any stream */
    _input_line_max = 25; /* set before 1st call */

    while( ( p = input_line( fp, buf, SIZ ) ) != NULL ) {
        printf( "%s\n", buf );
        fflush( stdout );
    }
    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

insque()

Insert an element into a doubly linked queue

Synopsis:

```
#include <search.h>

void insque( void *elem,
             void *pred);
```

Arguments:

elem

A pointer to the element you want to insert.

pred

A pointer to the previous element in the list, or NULL if you want to initialize the head of the list.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *insque()* function inserts the element pointed to by *elem* into a doubly linked queue immediately after the element pointed to by *pred*. The queue can be either circular or linear.

The first two members of the elements must be pointers to the same type of structure; the names of the members don't matter. The first member of the structure is a forward pointer to the next element in the queue, and the second is a backward pointer to the previous element. The application can define any additional members in the structure.

If the queue is linear, the queue is terminated with **NULL** pointers.

If the queue is to be used as a linear list, invoking `insque( &element, NULL)`, where *element* is the initial element of the queue, initializes the forward and backward pointers of the element to `NULL`.

If you intend to use the queue as a circular list, initialize the forward pointer and the backward pointer of the initial element of the queue to the element's own address.

You can use [remque\(\)](#) (p. 2559) to remove elements from the queue.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptAttach(), InterruptAttach_r()

Attach an interrupt handler to an interrupt source

Synopsis:

```
#include <sys/neutrino.h>

int InterruptAttach( int intr,
                    const struct sigevent * (* handler)(void *, int),
                    const void * area,
                    int size,
                    unsigned flags );

int InterruptAttach_r( int intr,
                      const struct sigevent * (* handler)(void *, int),
                      const void * area,
                      int size,
                      unsigned flags );
```

Arguments:

intr

The interrupt that you want to attach a handler to; see “[Interrupt vector numbers](#) (p. 1148),” below.

handler

A pointer to the handler function; see “[Interrupt handler function](#) (p. 1150),” below.

area

A pointer to a communications area in your process, or NULL if you don't want a communications area.

size

The size of the communications area; this should be 0 if *area* is NULL. *InterruptAttach()* currently ignores this argument.

flags

Flags that specify how you want to attach the interrupt handler. For more information, see “[Flags](#) (p. 1151),” below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptAttach()` and `InterruptAttach_r()` kernel calls attach the interrupt function *handler* to the hardware interrupt specified by *intr*. They automatically enable (i.e., unmask) the interrupt level.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1152) section for details.

Before calling either of these functions, the thread must:

- have the `PROCMGR_AID_INTERRUPT` and `PROCMGR_AID_IO` abilities enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do all of the above, the attachment fails with an error code of `EPERM`.

On a multicore system, the interrupt handler runs on the CPU that takes the interrupt.

Interrupt vector numbers

The interrupt values for *intr* are *logical interrupt vector numbers* grouped into related “interrupt classes” that generally correspond to a particular interrupt line on the CPU. The following interrupt classes are present on all QNX Neutrino systems:

`_NTO_INTR_CLASS_EXTERNAL`

Normal external interrupts (such as the ones generated by the `INTR` pin on x86 CPUs).

`_NTO_INTR_CLASS_SYNTHETIC`

Synthetic, kernel-generated interrupts.

`_NTO_INTR_SPARE` is usually the only `_NTO_INTR_CLASS_SYNTHETIC` interrupt you'll use; `_NTO_INTR_SPARE` is guaranteed not to match any valid logical interrupt vector number.

There can be additional interrupt classes defined for specific CPUs or embedded systems. For the interrupt assignments for specific boards, see:

- some `*intr.h` files in the platform-specific directories under `${QNX_TARGET}/usr/include`
- the sample buildfiles in `${QNX_TARGET}/x86/boot/build`
- the buildfiles in the Board Support Package for your board

Interrupts and startup code

The mapping of logical interrupt vector numbers is completely dependent on the implementer of the startup code.

Device drivers must:

- Let the user specify an interrupt number on the command line; don't use a hard-coded value. Eventually, the configuration manager will provide interrupt numbers for the device drivers.
- Store interrupt numbers in an unsigned `int` variable; don't assume an interrupt number fits into a byte.

Typical x86 Interrupt vector numbers

The following list contains typical interrupt assignments for the 16 hardware interrupts on an x86-based PC using `startup-bios`:

Interrupt <i>intr</i>	Description
0	A clock that runs at the resolution set by ClockPeriod() (p. 329)
1	Keyboard
2	Slave 8259 — you can't attach to this interrupt.
3	Com2
4	Com1
5	Net card / sound card / other
6	Floppy
7	Parallel printer / sound card / other
8	
9	Remapped interrupt 2
10	
11	
12	
13	Co-processor
14	Primary disk controller
15	Secondary disk controller



The interrupt assignments are different for other boards.

Interrupt handler function

The function to call is specified by the *handler* argument. This function runs in the environment of your process, and the *area* and *size* arguments define a communications area in your process. This typically is a structure containing buffers and information needed by the *handler* and the process when it runs.



The *area* argument can be `NULL` to indicate no communications area. If *area* is `NULL`, *size* should be 0.

The *handler* function's prototype is:

```
const struct sigevent* handler( void* area, int id );
```

where *area* is a pointer to the *area* specified by the call to *InterruptAttach()*, and *id* is the ID returned by *InterruptAttach()*.

Follow the following guidelines when writing your handler:

- A temporary interrupt stack of limited depth is provided at interrupt time, so avoid placing large arrays or structures on the stack frame of the handler. It's safe to assume that about 200 bytes of stack are available.
- The interrupt handler runs asynchronously with the threads in the process. Any variables modified by the handler should be declared with the `volatile` keyword and modified with interrupts disabled or using the *atomic**(*)* functions in any thread and ISR.
- The interrupt handler should be kept as short as possible. If a significant amount of work needs to be done, the handler should deliver an event to awaken a thread to do the work.
- The handler can't call library routines that contain kernel calls *except* for [InterruptDisable\(\)](#) (p. 1164), [InterruptEnable\(\)](#) (p. 1166), [InterruptLock\(\)](#) (p. 1174), [InterruptMask\(\)](#) (p. 1176), [InterruptUnlock\(\)](#) (p. 1179), and [InterruptUnmask\(\)](#) (p. 1181).

The handler can call [TraceEvent\(\)](#) (p. 3510), but not all of its commands are valid.

- It isn't safe to use floating-point operations in Interrupt Service Routines.

The return value of the *handler* function should be `NULL` or a pointer to a valid [sigevent](#) (p. 2961) structure that the kernel delivers. These events are defined in `<signal.h>`.

Consider the following when choosing the event type:

- Message-driven processes that block in a receive loop using [MsgReceivev\(\)](#) (p. 1771) should consider using `SIGEV_PULSE` to trigger a pulse.

- Threads that block at a particular point in their code and don't go back to a common receive point should consider using `SIGEV_INTR` as the event notification type and [InterruptWait\(\)](#) (p. 1183) as the blocking call.



The thread that calls *InterruptWait()* *must* be the one that called *InterruptAttach()*.

- Using `SIGEV_SIGNAL`, `SIGEV_SIGNAL_CODE`, `SIGEV_SIGNAL_THREAD`, or `SIGEV_THREAD` is discouraged. It's less efficient than the other mechanisms for interrupt event delivery.

Flags

The *flags* argument is a bitwise OR of the following values, or 0:

Flag	Description
_NTO_INTR_FLAGS_END (p. 1151)	Put the new handler at the end of the list of existing handlers (for shared interrupts) instead of the start.
_NTO_INTR_FLAGS_PROCESS (p. 1151)	Associate the handler with the process instead of the attaching thread.
_NTO_INTR_FLAGS_TRK_MSK (p. 1152)	Track calls to InterruptMask() (p. 1176) and InterruptUnmask() (p. 1181) to make detaching the interrupt handler safer.

`_NTO_INTR_FLAGS_END`

The interrupt structure allows hardware interrupts to be shared. For example, if two processes take over the same physical interrupt, both handlers are invoked consecutively. When a handler attaches, it's placed in front of any existing handlers for that interrupt and is called first. You can change this behavior by setting the `_NTO_INTR_FLAGS_END` flag in the *flags* argument. This adds the handler at the end of any existing handlers. Although the microkernel allows full interrupt sharing, your hardware might not. For example, the ISA bus doesn't allow interrupt sharing, while the PCI bus does.

Processor interrupts are enabled during the execution of the handler. *Don't* attempt to talk to the interrupt controller chip. The operating system issues the end-of-interrupt command to the chip after processing all handlers at a given level.

The first process to attach to an interrupt unmask the interrupt. When the last process detaches from an interrupt, the system masks it.

If the thread that attached the interrupt handler terminates without detaching the handler, the kernel does it automatically.

`_NTO_INTR_FLAGS_PROCESS`

Adding `_NTO_INTR_FLAGS_PROCESS` to *flags* associates the interrupt handler with the *process* instead of the attaching thread. The interrupt handler is removed when the process exits, instead of when the attaching thread exits.

`_NTO_INTR_FLAGS_TRK_MSK`

The `_NTO_INTR_FLAGS_TRK_MSK` flag and the *id* argument to *InterruptMask()* and *InterruptUnmask()* let the kernel track the number of times a particular interrupt handler or event has been masked. Then, when an application detaches from the interrupt, the kernel can perform the proper number of unmask to ensure that the interrupt functions normally. This is important for shared interrupt levels.



You should always set
`_NTO_INTR_FLAGS_TRK_MSK`.

Blocking states

This call doesn't block.

Returns:

The only difference between these functions is the way they indicate errors:

InterruptAttach()

An interrupt function ID. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

InterruptAttach_r()

An interrupt function ID. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Use the function ID with the [InterruptDetach\(\)](#) (p. 1162) function to detach this interrupt handler.

Errors:

EAGAIN

All kernel interrupt entries are in use.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The value of *intr* isn't a valid interrupt number.

EPERM

The process doesn't have I/O privileges, or it doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

If you're writing a resource manager and using the *resmgr_**() functions with multiple threads, a thread that attaches to an interrupt *must* use [_NTO_INTR_FLAGS_PROCESS](#) (p. 1151) in the *flags* argument when calling *InterruptAttach()*.

If your interrupt handler isn't SMP-safe, you must lock it to one processor using:

```
ThreadCtl( _NTO_TCTL_RUNMASK, ... );
```

InterruptAttachEvent(), InterruptAttachEvent_r()

Attach an event to an interrupt source

Synopsis:

```
#include <sys/neutrino.h>

int InterruptAttachEvent(
    int intr,
    const struct sigevent* event,
    unsigned flags );

int InterruptAttachEvent_r(
    int intr,
    const struct sigevent* event,
    unsigned flags );
```

Arguments:

intr

The *interrupt vector number* that you want to attach an event to; for more information, see “[Interrupt vector numbers](#) (p. 1148)” in the documentation for [InterruptAttach\(\)](#) (p. 1147).

event

A pointer to the [sigevent](#) (p. 2961) structure that you want to be delivered when this interrupt occurs.

flags

Flags that specify how you want to attach the interrupt handler. For more information, see “[Flags](#) (p. 1155),” below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *InterruptAttachEvent()* and *InterruptAttachEvent_r()* kernel calls attach the given event to the hardware interrupt specified by *intr*. They automatically enable (i.e., unmask) the interrupt level.

The *InterruptAttachEvent()* and *InterruptAttachEvent_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1157) section for details.

Before calling either of these functions, the thread must:

- have the `PROCmgr_AID_INTERRUPT` and `PROCmgr_AID_IO` abilities enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't have the appropriate abilities, the call to *InterruptAttachEvent()* fails with an error of `EPERM`; if the thread doesn't successfully call *ThreadCtl()*, it might `SIGSEGV` when it calls *InterruptAttachEvent()* or *InterruptAttachEvent_r()*.

To prevent infinite interrupt recursion, the kernel automatically does an [InterruptMask\(\)](#) (p. 1176) for *intr* when delivering the event. After the interrupt-handling thread has dealt with the event, it must call [InterruptUnmask\(\)](#) (p. 1181) to reenable the interrupt.

Consider the following when choosing an event type:

- Message-driven processes that block in a receive loop using [MsgReceivev\(\)](#) (p. 1771) should consider using `SIGEV_PULSE` to trigger a channel.
- Threads that block at a particular point in their code and don't go back to a common receive point, should consider using `SIGEV_INTR` as the event notification type and [InterruptWait\(\)](#) (p. 1183) as the blocking call.



The thread that calls *InterruptWait()* must be the one that called *InterruptAttachEvent()*.

- Using `SIGEV_SIGNAL`, `SIGEV_SIGNAL_CODE`, or `SIGEV_SIGNAL_THREAD` is discouraged. It is less efficient than the other mechanisms for interrupt event delivery.

On a multicore system, the thread that receives the event set up by *InterruptAttachEvent()* runs on any CPU, limited only by the scheduler and the runmask.

Flags

The *flags* argument is a bitwise OR of the following values, or 0:

Flag	Description
_NTO_INTR_FLAGS_END (p. 1156)	Put the new event at the end of the list of existing events instead of the start.
_NTO_INTR_FLAGS_PROCESS (p. 1156)	Associate the event with the process instead of the attaching thread.

Flag	Description
<code>_NTO_INTR_FLAGS_TRK_MSK</code> (p. 1156)	Track calls to InterruptMask() (p. 1176) and InterruptUnmask() (p. 1181) to make detaching the interrupt handler safer.

`_NTO_INTR_FLAGS_END`

The interrupt structure allows hardware interrupts to be shared. For example if two processes call *InterruptAttachEvent()* for the same physical interrupt, both events are sent consecutively. When an event attaches, it's placed in front of any existing events for that interrupt and is delivered first. You can change this behavior by setting the `_NTO_INTR_FLAGS_END` flag in the *flags* argument. This adds the event at the end of any existing events.

`_NTO_INTR_FLAGS_PROCESS`

Adding `_NTO_INTR_FLAGS_PROCESS` to *flags* associates the interrupt event with the *process* instead of the attaching thread. The interrupt event is removed when the process exits, instead of when the attaching thread exits.



The kernel automatically attempts to set the `_NTO_INTR_FLAGS_PROCESS` flag if the event is directed at the process in general (for `SIGEV_SIGNAL`, `SIGEV_SIGNAL_CODE`, and `SIGEV_PULSE` events).

`_NTO_INTR_FLAGS_TRK_MSK`

The `_NTO_INTR_FLAGS_TRK_MSK` flag and the *id* argument to *InterruptMask()* and *InterruptUnmask()* let the kernel track the number of times a particular interrupt handler or event has been masked. Then, when an application detaches from the interrupt, the kernel can perform the proper number of unmask to ensure that the interrupt functions normally. This is important for shared interrupt levels.



You should always set `_NTO_INTR_FLAGS_TRK_MSK`.

Advantages & disadvantages

InterruptAttachEvent() has several advantages over [InterruptAttach\(\)](#) (p. 1147):

- Less work is done at interrupt time (you avoid the context switch necessary to map in an interrupt handler).
- Interrupt handling code runs at the thread's priority, which lets you specify the priority of the interrupt handling.
- You can use process-level debugging on your interrupt handler code.
- It isn't safe to use floating-point operations in Interrupt Service Routines.

There are also some disadvantages:

- There might be a delay before the interrupt handling code runs (until the thread is scheduled to run).
- For multiple devices sharing an event, the amount of time spent with the interrupt masked increases.

You can freely mix calls to [InterruptAttach\(\)](#) (p. 1147) and *InterruptAttachEvent()* for a particular interrupt.

Blocking states

This call doesn't block.

Returns:

The only difference between these functions is the way they indicate errors:

InterruptAttachEvent()

An interrupt function ID. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

InterruptAttachEvent_r()

An interrupt function ID. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Use the ID with [InterruptDetach\(\)](#) (p. 1162) to detach this interrupt event.

Errors:

EAGAIN

All kernel interrupt entries are in use.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The value of *intr* isn't a valid interrupt number.

EPERM

The process doesn't have I/O privileges, or it doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

InterruptCharacteristic(), InterruptCharacteristic_r()

Get or set a characteristic associated with an interrupt

Synopsis:

```
#include <sys/neutrino.h>

int InterruptCharacteristic( int type,
                           int id,
                           unsigned *new,
                           unsigned *old );

int InterruptCharacteristic_r( int type,
                              int id,
                              unsigned *new,
                              unsigned *old );
```

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Arguments:

type

The type of characteristic; the only type that's currently defined is:

- `_NTO_IC_LATENCY` — set or get the interrupt's latency.

id

The identifier for the interrupt function, as returned by [InterruptAttach\(\)](#) (p. 1147) or [InterruptAttachEvent\(\)](#) (p. 1154).

new

NULL, or a pointer to the new value.

old

NULL, or a pointer to a location where the function can store the old value.

Description:

The *InterruptCharacteristic()* and *InterruptCharacteristic_r()* kernel calls set or get a characteristic of an interrupt.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1160) section for details.

The only *type* currently defined is `_NTO_IC_LATENCY`. If the *new* argument is non-NULL, it points to an `unsigned` that contains the new latency value for the interrupt, in nanoseconds. If the *old* argument is non-NULL, it's filled in with the old latency value for the interrupt. The default latency value is zero. In order to set a latency, the calling thread must be the process that attached to the interrupt, and it must have obtained I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

The *id* argument can be -1. In this case, you're setting the global latency value for the system. If an interrupt attachment doesn't specify a latency value, the global latency number is used when calculating how deep a sleep state the kernel may use. You need to have I/O privileges to set the global latency value.

Returns:

The only difference between these functions is the way they indicate errors:

InterruptCharacteristic()

0 on success. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

InterruptCharacteristic_r()

0 on success. This function does **NOT** set *errno*. If an error occurs, a value from the Errors section is returned.

Errors:

EFAULT

The function couldn't read from the *new* pointer, or it couldn't write to the *old*.

EINVAL

The *type* isn't `_NTO_IC_LATENCY`.

EPERM

You're trying to set a new latency, but the calling thread isn't the attaching process or doesn't have I/O privileges.

ESRCH

The *id* isn't either a valid interrupt attachment identifier or -1.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptDetach(), InterruptDetach_r()

Detach an interrupt handler by ID

Synopsis:

```
#include <sys/neutrino.h>

int InterruptDetach( int id );
int InterruptDetach_r( int id );
```

Arguments:

id

The value returned by [InterruptAttach\(\)](#) (p. 1147) [InterruptAttachEvent\(\)](#) (p. 1154), or [InterruptHookIdle\(\)](#) (p. 1168).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls detach the interrupt handler specified by the *id* argument. If, after detaching, no thread is attached to the interrupt, then the interrupt is masked off. The thread that detaches the interrupt handler must be in the same process as the thread that attached it.

The *InterruptDetach()* and *InterruptDetach_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1163) section for details.

Before calling either of these functions, the thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do these things, it might `SIGSEGV` when it calls *InterruptDetach()* or *InterruptDetach_r()*.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

InterruptDetach()

If an error occurs, -1 is returned and *errno* (p. 539) is set. Any other value returned indicates success.

InterruptDetach_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EINVAL

The value of *id* doesn't exist for this process.

EPERM

The process doesn't have superuser privileges, or isn't the process that attached the interrupt identified by *id*.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

InterruptDisable()

Disable hardware interrupts

Synopsis:

```
#include <sys/neutrino.h>

void InterruptDisable( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *InterruptDisable()* function disables all hardware interrupts. You can call it from a thread or from an interrupt handler. Before calling this function, the thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NT0_TCTL_IO, 0 );
```



Any kernel call results in the re-enabling of interrupts, and many library routines are built on kernel calls. Masked interrupts aren't affected.

If the thread doesn't do these things, it might `SIGSEGV` when it calls *InterruptDisable()*.

Reenable the interrupts by calling [InterruptEnable\(\)](#) (p. 1166).



Since this function disables all hardware interrupts, take care to reenable them as quickly as possible. Failure to do so may result in increased interrupt latency and nonrealtime performance.

Use *InterruptDisable()* instead of an inline `cli` to ensure hardware portability with non-x86 CPUs.



Use [InterruptLock\(\)](#) (p. 1174) and [InterruptUnlock\(\)](#) (p. 1179) instead of *InterruptDisable()* and [InterruptEnable\(\)](#) (p. 1166). The *InterruptLock()* and *InterruptUnlock()* functions perform the intended function on SMP hardware, and allow your interrupt thread to run on any processor in the system.

Classification:[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptEnable()

Enable hardware interrupts

Synopsis:

```
#include <sys/neutrino.h>

void InterruptEnable( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *InterruptEnable()* function enables all hardware interrupts. You can call it from a thread or from an interrupt handler. Before calling this function, the thread must:

- have the PROCmgr_AID_IO ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NT0_TCTL_IO, 0 );
```

If the thread doesn't do these things, it might SIGSEGV when it calls *InterruptEnable()*. You should call this function as quickly as possible after calling [InterruptDisable\(\)](#) (p. 1164).



Use [InterruptLock\(\)](#) (p. 1174) and [InterruptUnlock\(\)](#) (p. 1179) instead of [InterruptDisable\(\)](#) (p. 1164) and *InterruptEnable()*. The *InterruptLock()* and *InterruptUnlock()* functions perform the intended function on SMP hardware, and allow your interrupt thread to run on any processor in the system.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

InterruptHookIdle()

Attach an “idle” interrupt handler

Synopsis:

```
#include <sys/neutrino.h>

int InterruptHookIdle(
    void (*handler)(uint64_t *, struct qtime_entry *),
    unsigned flags );
```

Arguments:

handler

A pointer to the handler function; see below.

flags

Flags that specify how you want to attach the interrupt handler. For more information, see “[Flags](#) (p. 1169),” below.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *InterruptHookIdle()* kernel call attaches the specified interrupt *handler* to the “idle” interrupt, which is called when the system is idle. This is typically used to implement power management features.



In order to attach the handler, your process must have the PROC_MGR_AID_INTERRUPT ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141) The thread doesn't need to have I/O privileges.

The arguments to the *handler* functions are:

uint64_t*

A pointer to the time, in nanoseconds, when the next timer will expire.

struct qtime_entry *

A pointer to the section of the system page with the time information, including the current time of day.

The simplest idle handler consists of a `halt` instruction.

Flags

The *flags* argument is a bitwise OR of the following values, or 0:

Flag	Description
_NTO_INTR_FLAGS_END (p. 1169)	Put the new handler at the end of the list of existing handlers (for shared interrupts) instead of the start.
_NTO_INTR_FLAGS_PROCESS (p. 1169)	Associate the handler with the process instead of the attaching thread.
_NTO_INTR_FLAGS_TRK_MSK (p. 1169)	Track calls to InterruptMask() (p. 1176) and InterruptUnmask() (p. 1181) to make detaching the interrupt handler safer.

`_NTO_INTR_FLAGS_END`

The interrupt structure allows hardware interrupts to be shared. For example, if two processes take over the same physical interrupt, both handlers are invoked consecutively. When a handler attaches, it's placed in front of any existing handlers for that interrupt and is called first. You can change this behavior by setting the `_NTO_INTR_FLAGS_END` flag in the *flags* argument. This adds the handler at the end of any existing handlers.

Processor interrupts are enabled during the execution of the handler. *Don't* attempt to talk to the interrupt controller chip. The end of interrupt command is issued to the chip by the operating system after processing all handlers at a given level.

The first process to attach to an interrupt unmask the interrupt. When the last process detaches from an interrupt, the system masks it.

If the thread that attached the interrupt handler terminates without detaching the handler, the kernel does it automatically.

`_NTO_INTR_FLAGS_PROCESS`

Adding `_NTO_INTR_FLAGS_PROCESS` to *flags* associates the interrupt handler with the *process* instead of the attaching thread. The interrupt handler is removed when the process exits, instead of when the attaching thread exits.

`_NTO_INTR_FLAGS_TRK_MSK`

The `_NTO_INTR_FLAGS_TRK_MSK` flag and the *id* argument to [InterruptMask\(\)](#) and [InterruptUnmask\(\)](#) let the kernel track the number of times a particular interrupt

handler or event has been masked. Then, when an application detaches from the interrupt, the kernel can perform the proper number of unmask to ensure that the interrupt functions normally. This is important for shared interrupt values.

Blocking states

This call doesn't block.

Returns:

An interrupt function ID, or -1 if an error occurs ([errno](#) (p. 539) is set).

Use the returned ID with the [InterruptDetach\(\)](#) (p. 1162) function to detach this interrupt handler.

Errors:

EAGAIN

All kernel interrupt entries are in use.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

InterruptHookTrace()

Attach the pseudo interrupt handler that the instrumented module uses

Synopsis:

```
#include <sys/neutrino.h>

int InterruptHookTrace(
    const struct sigevent * (* handler)(int),
    unsigned flags );
```

Arguments:

handler

A pointer to the handler function.

flags

Flags that specify how you want to attach the interrupt handler; 0, or the following bit:

- [_NTO_INTR_FLAGS_END](#) (p. 1172) — put the new handler at the end of the list of existing handlers (for shared interrupts) instead of the start. For more information, see below.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *InterruptHookTrace()* kernel call attaches the pseudo interrupt handler *handle* that the instrumented module uses.



This function requires the instrumented kernel. For more information, see the System Analysis Toolkit (SAT) *User's Guide*.

Before calling this function, the thread must:

- have the `PROCmgr_AID_INTERRUPT` and `PROCmgr_AID_IO` abilities enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

The *handler* argument specifies the pseudo interrupt handler that receives trace events from the kernel. The integer that's passed to the handler is a combination of the buffer index and the sequence number; to extract each part, pass the integer to the `_TRACE_GET_BUFFNUM()` and `_TRACE_GET_BUFFSEQ()` macros defined in `<sys/trace.h>`.

`_NTO_INTR_FLAGS_END`

The interrupt structure allows trace interrupts to be shared. For example, if two processes take over the same trace interrupt, both handlers are invoked consecutively. When a handler attaches, it's placed in front of any existing handlers for that interrupt and is called first. This behavior can be changed by setting the `_NTO_INTR_FLAGS_END` flag in the *flags* argument. This adds the handler at the end of any existing handlers.

Blocking states

This call doesn't block.

Returns:

An interrupt function ID, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EAGAIN

All kernel interrupt entries are in use.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EPERM

The process doesn't have I/O privileges, or it doesn't have the required permission; see *procmgr_ability()* (p. 2141).

ENOTSUP

The kernel isn't instrumented.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

InterruptLock()

Guard a critical section in an interrupt handler

Synopsis:

```
#include <sys/neutrino.h>

void InterruptLock( intrspin_t* spinlock );
```

Arguments:

spinlock

The *spinlock* (a variable shared between the interrupt handler and a thread) to use.

If *spinlock* isn't a static variable, you must initialize it by calling:



```
memset( spinlock, 0, sizeof( *spinlock ) );
```

before using it with *InterruptLock()*.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *InterruptLock()* function guards a critical section by locking the specified *spinlock*. You can call this function from a thread or from an interrupt handler. Before calling this function, the thread must:

- have the PROCmgr_AID_IO ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do these things, it might SIGSEGV when it calls *InterruptLock()*.

This function tries to acquire the *spinlock* (a variable shared between the interrupt handler and a thread) while interrupts are disabled. The code spins in a tight loop until the lock is acquired. It's important to release the lock as soon as possible. Typically, this is a few lines of code without any loops:

```
InterruptLock( &spinner );
/* ... critical section */
InterruptUnlock( &spinner );
```

InterruptLock() solves a common need in many realtime systems to protect access to shared data structures between an interrupt handler and the thread that owns the handler. The traditional POSIX primitives used between threads aren't available for use by an interrupt handler.

The *InterruptLock()* and *InterruptUnlock()* functions work on single-processor or multiprocessor machines.



Any kernel call results in the re-enabling of interrupts, and many library routines are built on kernel calls. Masked interrupts are not affected.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptMask()

Disable a hardware interrupt

Synopsis:

```
#include <sys/neutrino.h>

int InterruptMask( int intr,
                  int id );
```

Arguments:

intr

The interrupt you want to mask.

id

The value returned by [InterruptAttach\(\)](#) (p. 1147) or [InterruptAttachEvent\(\)](#) (p. 1154), or -1 if you don't want the kernel to track interrupt maskings and unmaskings for each handler.



The *id* is ignored unless you use the `_NTO_INTR_FLAGS_TRK_MSK` flag when you attach the handler.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *InterruptMask()* kernel call disables the hardware interrupt specified by *intr* for the handler specified by *id*. You can call this function from a thread or from an interrupt handler. Before calling this function, the thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do these things, it might `SIGSEGV` when it calls *InterruptMask()*.

Reenable the interrupt by calling [InterruptUnmask\(\)](#) (p. 1181).

The kernel automatically enables an interrupt when the first handler attaches to it using [InterruptAttach\(\)](#) (p. 1147) and disables it when the last handler detaches.

This call is often used when a device presents a level-sensitive interrupt to the system that can't be easily cleared in the interrupt handler. Since the interrupt is level-sensitive, you can't exit the handler with the interrupt line active and unmasked. [InterruptMask\(\)](#) lets you mask the interrupt in the handler and schedule a thread to do the real work of communicating with the device to clear the source. Once cleared, the thread should call [InterruptUnmask\(\)](#) to reenabte this interrupt.

To disable all hardware interrupts, use the [InterruptLock\(\)](#) (p. 1174) function.



To ensure hardware portability, use [InterruptMask\(\)](#) instead of writing code that talks directly to the interrupt controller.

Calls to [InterruptMask\(\)](#) are nested; the interrupt isn't unmasked until [InterruptUnmask\(\)](#) (p. 1181) has been called once for every call to [InterruptMask\(\)](#).

Returns:

The current mask level count for success; or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of *intr* isn't a supported hardware interrupt.

ESRCH

The *id* parameter isn't something returned by [InterruptAttach\(\)](#) (p. 1147) or [InterruptAttachEvent\(\)](#) (p. 1154), or -1.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptStatus()

Determine whether or not interrupts are enabled

Synopsis:

```
#include <sys/neutrino.h>

unsigned InterruptStatus( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *InterruptStatus()* function indicates whether or not interrupts are currently enabled.

Returns:

0 if interrupts are disabled, or a nonzero value (not necessarily 1) if they're enabled.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptUnlock()

Release a critical section in an interrupt handler

Synopsis:

```
#include <sys/neutrino.h>

void InterruptUnlock( intrspin_t* spinlock );
```

Arguments:

spinlock

The spinlock (a variable shared between the interrupt handler and a thread) used in a call to [InterruptLock\(\)](#) (p. 1174) to lock the handler.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *InterruptUnlock()* function releases a critical section by unlocking the specified *spinlock*, reenabling interrupts. You can call this function from a thread or from an interrupt handler.

Before calling this function, the thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do these things, it might `SIGSEGV` when it calls *InterruptUnlock()*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptUnmask()

Enable a hardware interrupt

Synopsis:

```
#include <sys/neutrino.h>

int InterruptUnmask( int intr,
                    int id );
```

Arguments:

intr

The interrupt you want to unmask.

id

The value returned by [InterruptAttach\(\)](#) (p. 1147) or [InterruptAttachEvent\(\)](#) (p. 1154), or -1 if you don't want the kernel to track interrupt maskings and unmaskings for each handler.



The *id* is ignored unless you use the `_NTO_INTR_FLAGS_TRK_MSK` flag when you attach the handler.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptUnmask()` kernel call enables the hardware interrupt specified by *intr* for the interrupt handler specified by *intr* for the handler specified by *id* when the mask count reaches zero. You can call this function from a thread or from an interrupt handler. Before calling this function, the thread must:

- have the `PROCmgr_AID_IO` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do these things, it might SIGSEGV when it calls *InterruptUnmask()*.

Calls to *InterruptMask()* (p. 1176) are nested; the interrupt isn't unmasked until *InterruptUnmask()* has been called once for every call to *InterruptMask()*.

Returns:

The current mask count, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EINVAL**

Not a supported hardware interrupt *intr*.

ESRCH

The *id* parameter isn't something returned by *InterruptAttach()* (p. 1147) or *InterruptAttachEvent()* (p. 1154), or -1.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

InterruptWait(), InterruptWait_r()

Wait for a hardware interrupt

Synopsis:

```
#include <sys/neutrino.h>

int InterruptWait( int flags,
                  const uint64_t * timeout );

int InterruptWait_r( int flags,
                    const uint64_t * timeout );
```

Arguments:

flags

This should currently be 0.

timeout

This should currently be NULL. This may change in future versions.



Use [TimerTimeout\(\)](#) (p. 3464) to achieve a timeout.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

These kernel calls wait for a hardware interrupt. The calling thread should have attached a handler to the interrupt, by calling [InterruptAttach\(\)](#) (p. 1147) or [InterruptAttachEvent\(\)](#) (p. 1154). The call to [InterruptWait\(\)](#) or [InterruptWait_r\(\)](#) blocks waiting for an interrupt handler to return an event with notification type SIGEV_INTR (i.e. a hardware interrupt).

The [InterruptWait\(\)](#) and [InterruptWait_r\(\)](#) functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1184) section for details.

If the notification event occurs before [InterruptWait\(\)](#) is called, a pending flag is set. When [InterruptWait\(\)](#) is called, the flag is checked; if set, it's cleared and the call immediately returns with success.

On a multicore system, a thread that calls *InterruptWait()* runs on any CPU, limited only by the scheduler and the runmask.

Blocking states

STATE_INTR

The thread is waiting for an interrupt handler to return a SIGEV_INTR event.

Returns:

The only difference between these functions is the way they indicate errors:

InterruptWait()

If an error occurs, -1 is returned and *errno* (p. 539) is set. Any other value returned indicates success.

InterruptWait_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EINTR

The call was interrupted by a signal.

ENOTSUP

The reserved arguments aren't NULL.

ETIMEDOUT

A kernel timeout unblocked the call. See *TimerTimeout()* (p. 3464).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_intr_v86()

Execute a real-mode software interrupt

Synopsis:

```
#include <x86/v86.h>

int _intr_v86( int swi,
               struct _v86reg* regs,
               void* data,
               int datasize );
```

Arguments:

swi

The software interrupt that you want to execute.

regs

A pointer to a `_v86reg` structure that specifies the values you want to use for the registers on entry to real mode; see below.

data

A pointer to the data that you want to copy into memory; see below.

datasize

The size of the data, in bytes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_intr_v86()` function executes the real-mode software interrupt specified by `swi` in virtual 8086 mode. This allows access to the ROM BIOS functions that are designed to run in 16-bit real mode. Two common examples are:

Interrupt	Description
<code>int 10h</code>	Video BIOS

Interrupt	Description
int 1ah	PCI

BIOS calls (such as int 13h, disk I/O) that require hardware interrupts to be directed at their code aren't supported.

Upon entry to real mode, the registers are loaded from *regs*. The segment registers and any pointers should address a 2 KB communication area located at offset 0:800h in real memory. The buffer *data* of length *datasize* is copied to this area just before real mode is entered and copied back when the call completes. At this point *regs* is also updated to contain the values of the real-mode registers.

You should set the DS, ES, FS and GS segment registers to 0. The values in the CS:IP, and SS:SP registers are ignored and are set by the kernel. The stack provided is about 500 bytes in size.

The layout of real mode memory is described by the structure `_v86_memory` in `<x86/v86.h>`.

When a thread enters virtual 8086 mode, all threads in the system continue to be scheduled based upon their priority, including the calling thread. While in virtual 8086 mode, full access to IO ports and interrupt enable and disable are allowed. Only one thread may enter virtual 8086 mode at a time.

This function fails if the calling process doesn't have the `PROCmgr_AID_V86` enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EBADF

The connection to the system process is no longer connected to a channel, or the connection doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when accessing the information pointed to by the *data* or *regs* arguments.

EINTR

The call was interrupted by a signal.

E_OVERFLOW

The sum of the IOV lengths being sent to the system process exceeds INT_MAX.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ETIMEDOUT

A kernel timeout unblocked the underlying call to [MsgSendvnc\(\)](#) (p. 1805).
See [TimerTimeout\(\)](#) (p. 3464).

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <errno.h>
#include <x86/v86.h>

struct _v86reg reg;

int main( void ) {
    char buf[4];

    /* Equipment call */
    printf("int 12\n");
    memset(&reg, 0, sizeof(reg));
    _intr_v86(0x12, &reg, NULL, 0);
    printreg();
    sleep(5);

    /* Enter 40 column text mode */
    printf("int 10 ah=00h al=00h\n");
    memset(&reg, 0, sizeof(reg));
    _intr_v86(0x10, &reg, NULL, 0);
    printreg();
    sleep(5);

    /* Enter 80 column text mode */
    printf("int 10 ah=00h al=02h\n");
    memset(&reg, 0, sizeof(reg));
    reg.eax = 2;
    _intr_v86(0x10, &reg, NULL, 0);
    printreg();
    sleep(5);

    /* Write a string from memory */
    printf("int 10 ah=13h al=00h\n");
    strcpy(buf, "Hi!");
    memset(&reg, 0, sizeof(reg));
    reg.eax = 0x1300;
    reg.es = 0;
    reg.ebp = offsetof(struct _v86_memory, userdata);
    reg.ecx = strlen(buf);
    reg.edx = 0;
    reg.ebx = 0x0007;
    _intr_v86(0x10, &reg, buf, strlen(buf));
    printreg();
    sleep(5);
}
```

```
    return EXIT_SUCCESS;
}

printreg() {
    printf("eax=%-8x ebx=%-8x ecx=%-8x edx=%-8x\n",
        reg.eax, reg.ebx, reg.ecx, reg.edx);
    printf("esi=%-8x edi=%-8x ebp=%-8x esp=%-8x\n",
        reg.esi, reg.edi, reg.ebp, reg.esp);
    printf(" ds=%-8x es=%-8x fs=%-8x gs=%-8x\n",
        reg.ds, reg.es, reg.fs, reg.gs);
    printf("efl=%-8x\n\n",
        reg.efl);
}
```

Classification:

QNX Neutrino (x86 only) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_io_connect

Structure of a resource manager's connect message

Synopsis:

```
struct _io_connect {
    uint16_t      type;
    uint16_t      subtype;
    uint32_t      file_type;
    uint16_t      reply_max;
    uint16_t      entry_max;
    uint32_t      key;
    uint32_t      handle;
    uint32_t      ioflag;
    uint32_t      mode;
    uint16_t      sflag;
    uint16_t      access;
    uint16_t      zero;
    uint16_t      path_len;
    uint8_t       eflag;
    uint8_t       extra_type;
    uint16_t      extra_len;
    char          path[1];
};
```

Description:

The `_io_connect` structure is used to describe a connect message that a resource manager receives and sends.

The members include:

type

`_IO_CONNECT`

subtype

The type of connection that the message concerns; one of:

- `_IO_CONNECT_COMBINE` — combine with an I/O message.
- `_IO_CONNECT_COMBINE_CLOSE` — combine with I/O message and always close.
- `_IO_CONNECT_OPEN`
- `_IO_CONNECT_UNLINK`
- `_IO_CONNECT_RENAME`
- `_IO_CONNECT_MKNOD`
- `_IO_CONNECT_READLINK`
- `_IO_CONNECT_LINK`

- `_IO_CONNECT_RSVD_UNBLOCK` — place holder in the jump table.
- `_IO_CONNECT_MOUNT`

file_type

The file type; one of the following (defined in `<sys/ftype.h>`):

- `_FTYPE_ANY` — the path name can be anything.
- `_FTYPE_LINK` — reserved for the Process Manager.
- `_FTYPE_MOUNT` — receive mount requests on the path (*path* must be NULL).
- `_FTYPE_MQUEUE` — reserved for a message-queue manager.
- `_FTYPE_PIPE` — reserved for a pipe manager.
- `_FTYPE_SEM` — reserved for a semaphore manager.
- `_FTYPE_SHMEM` — reserved for a shared memory object.
- `_FTYPE_SOCKET` — reserved for a socket manager.
- `_FTYPE_SYMLINK` — reserved for the Process Manager.

reply_max

The maximum length of the reply message.

entry_max

The maximum number of `_io_connect_entry` structures that the resource manager is willing to accept. If a path could reference more than one resource manager, it returns a list of `_io_connect_entry` structures referring to the overlapping resource managers.

key

Reserved.

handle

The handle returned by [*resmgr_attach\(\)*](#) (p. 2580).

ioflag

The bottom two bits are modified from traditional Unix values to more useful bit flags:

- `O_RDONLY` (0) is converted to `_IO_FLAG_RD` (0x01)
- `O_WRONLY` (1) is converted to `_IO_FLAG_WR` (0x02)
- `O_RDWR` (2) is converted to `_IO_FLAG_RD | _IO_FLAG_WR` (0x03)

Note that this translation can be performed without overlapping other `O_*` flags.

To check for permissions, use the following:

- `ioflag & _IO_FLAG_RD` for read permission
- `ioflag & _IO_FLAG_WR` for write permission
- If the file is open for reading and writing, both bits are set.

The remaining values of `ioflag` are outside this range and are not modified. These values are:

- `O_APPEND` — if set, the file offset is set to the end of the file prior to each write.
- `O_CREAT` — create the file.
- `O_DSYNC` — if set, this flag affects subsequent I/O calls; each call to [write\(\)](#) (p. 3737) waits until all data is successfully transferred to the storage device such that it's readable on any subsequent open of the file (even one that follows a system failure) in the absence of a failure of the physical storage medium. If the physical storage medium implements a non-write-through cache, then a system failure may be interpreted as a failure of the physical storage medium, and data may not be readable even if this flag is set and the `write()` indicates that it succeeded.
- `O_EXCL` — if you set both `O_EXCL` and `O_CREAT`, `open()` fails if the file exists. The check for the existence of the file and the creation of the file if it doesn't exist are atomic; no other process that's attempting the same operation with the same filename at the same time will succeed. Specifying `O_EXCL` without `O_CREAT` has no effect.
- `O_LARGEFILE` — allow the file offset to be 64 bits long.
- `O_NOCTTY` — if set, and `path` identifies a terminal device, the `open()` function doesn't cause the terminal device to become the controlling terminal for the process.
- `O_NONBLOCK` — don't block.
- `O_REALIDS` — use the real `uid/gid` for permissions checking.
- `O_RSYNC` — read I/O operations on the file descriptor complete at the same level of integrity as specified by the `O_DSYNC` and `O_SYNC` flags.
- `O_SYNC` — if set, this flag affects subsequent I/O calls; each call to [read\(\)](#) (p. 2502) or [write\(\)](#) (p. 3737) is complete only when both the data has been successfully transferred (either read or written) and all file system information relevant to that I/O operation (including that required to retrieve said data) is successfully transferred, including file update and/or access times, and so on. See the discussion of a successful data transfer in `O_DSYNC`, above.

- **O_TRUNC** — if the file exists and is a regular file, and the file is successfully opened **O_WRONLY** or **O_RDWR**, the file length is truncated to zero and the mode and owner are left unchanged. **O_TRUNC** has no effect on FIFO or block or character special files or directories. Using **O_TRUNC** with **O_RDONLY** has no effect.

mode

Contains the type and access permissions of the file.

The type is one of:

- **S_IFBLK** — block special.
- **S_IFCHR** — character special.
- **S_IFDIR** — directory.
- **S_IFIFO** — FIFO special.
- **S_IFLNK** — symbolic link.
- **S_IFMT** — type of file.
- **S_IFNAM** — special named file.
- **S_IFREG** — regular.
- **S_IFSOCK** — socket.

The permissions are a combination of:

Owner	Group	Others	Permission
S_IRUSR	S_IRGRP	S_IROTH	Read
S_IRWXU	S_IRWXG	S_IRWXO	Read, write, execute/search. A bitwise inclusive OR of the other three constants (S_IRWXU is OR of IRUSR , S_IRWSUR and S_IXUSR .)
S_IWUSR	S_IWGRP	S_IWOTH	Write
S_IXUSR	S_IXGRP	S_IXOTH	Execute/search

The following bits define miscellaneous permissions used by other implementations:

Bit	Equivalent
S_IEXEC	S_IXUSR
S_IREAD	S_IRUSR
S_IWRITE	S_IWUSR

sflag

How the client wants the file to be shared; a combination of the following bits:

- SH_COMPAT — set compatibility mode.
- SH_DENYRW — prevent read or write access to the file.
- SH_DENYWR — prevent write access to the file.
- SH_DENYRD — prevent read access to the file.
- SH_DENYNO — permit both read and write access to the file.

access

Contains a combination of `_IO_FLAG_RD` and `_IO_FLAG_WR` bits, which are used internally as the access permissions to allow from `ioflag`.

path_len

The length of the *path* member.

eflag

Extended flags:

- `_IO_CONNECT_EFLAG_DIR` — the path referenced a directory.
- `_IO_CONNECT_EFLAG_DOT` — the last component of a path was `.` or `..` (i.e. the current or parent directory).

extra_type

One of:

- `_IO_CONNECT_EXTRA_NONE`
- `_IO_CONNECT_EXTRA_LINK`
- `_IO_CONNECT_EXTRA_SYMLINK`
- `_IO_CONNECT_EXTRA_MQUEUE`
- `_IO_CONNECT_EXTRA_SOCKET`
- `_IO_CONNECT_EXTRA_SEM`
- `_IO_CONNECT_EXTRA_RESMGR_LINK`

-
- `_IO_CONNECT_EXTRA_PROC_SYMLINK`
 - `_IO_CONNECT_EXTRA_RENAME`
 - `_IO_CONNECT_EXTRA_MOUNT`
 - `_IO_CONNECT_EXTRA_MOUNT_OCB`

extra_len

The length of any extra data included in the message.

path

The path that the client is trying to connect to, relative to the resource manager's mountpoint.

Classification:

[QNX Neutrino](#) (p. 46)

`_io_connect_ftype_reply`

Structure of a connect message giving a status and a file type

Synopsis:

```
struct _io_connect_ftype_reply {  
    uint16_t status;  
    uint16_t reserved;  
    uint32_t file_type;  
};
```

Description:

A resource manager uses the `_io_connect_ftype_reply` structure to send a status and a file type to a client that has sent a connect message.

The members include:

status

Typically one of the [errno](#) (p. 539) values.

file_type

The file type; one of the following (defined in `<sys/ftype.h>`):

- `_FTYPE_ANY` — the path name can be anything.
- `_FTYPE_LINK` — reserved for the Process Manager.
- `_FTYPE_MOUNT` — receive mount requests on the path (*path* must be NULL).
- `_FTYPE_MQUEUE` — reserved for a message-queue manager.
- `_FTYPE_PIPE` — reserved for a pipe manager.
- `_FTYPE_SEM` — reserved for a semaphore manager.
- `_FTYPE_SHMEM` — reserved for a shared memory object.
- `_FTYPE_SOCKET` — reserved for a socket manager.
- `_FTYPE_SYMLINK` — reserved for the Process Manager.

Classification:

[QNX Neutrino](#) (p. 46)

_io_connect_link_reply

Structure of a connect message that redirects a client to another resource

Synopsis:

```
struct _io_connect_link_reply {
    uint32_t          reserved1;
    uint32_t          file_type;
    uint8_t           eflag;
    uint8_t           reserved2[1];
    uint16_t          chroot_len;
    uint32_t          umask;
    uint16_t          nentries;
    uint16_t          path_len;
/*
    struct _io_connect_entry  server[nentries];
    char                     path[path_len];
or
    struct _server_info      info;
    io_?_t                   msg;
*/
};
```

Description:

A resource manager uses the `_io_connect_link_reply` structure in a reply to a client that redirects the client to another resource. The members include:

file_type

The file type; one of the following (defined in `<sys/ftype.h>`):

- `_FTYPE_ANY` — the path name can be anything.
- `_FTYPE_LINK` — reserved for the Process Manager.
- `_FTYPE_MOUNT` — receive mount requests on the path (*path* must be `NULL`).
- `_FTYPE_QUEUE` — reserved for a message-queue manager.
- `_FTYPE_PIPE` — reserved for a pipe manager.
- `_FTYPE_SEM` — reserved for a semaphore manager.
- `_FTYPE_SHMEM` — reserved for a shared memory object.
- `_FTYPE_SOCKET` — reserved for a socket manager.
- `_FTYPE_SYMLINK` — reserved for the Process Manager.

eflag

Extended flags:

- `_IO_CONNECT_EFLAG_DIR` — the path referenced a directory.

- `_IO_CONNECT_EFLAG_DOT` — the last component of a path was `.` or `..` (i.e. the current or parent directory).

chroot_len

The length of `chroot` in the returned path.

umask

One of:

- `S_IFBLK` — block special.
- `S_IFCHR` — character special.
- `S_IFDIR` — directory.
- `S_IFIFO` — FIFO special.
- `S_IFLNK` — symbolic link.
- `S_IFMT` — type of file.
- `S_IFNAM` — special named file.
- `S_IFREG` — regular.
- `S_IFSOCK` — socket.

nentries

If this member is zero, the path is a symbolic link.

path_len

The length of the path including the terminating null character. If this member is zero, the path is null-terminated.

Classification:

[QNX Neutrino](#) (p. 46)

ioctl(), *ioctl_socket()*

Control a device

Synopsis:

```
#include <sys/ioctl.h>

int ioctl( int fd,
           int request,
           ... );

int ioctl_socket( int fd,
                  int request,
                  ... );
```

Arguments:

fd

An open file descriptor for the file or device that you want to manipulate.

request

What you want to do to the file or device; see below for a summary. The macros and definitions that you use in specifying a request are located in the file `<sys/ioctl.h>`.

Additional arguments

As required by the request.

Library:

Function	Library	Linking option to gcc
<i>ioctl()</i>	libc	-l c (This library is usually included automatically.)
<i>ioctl_socket()</i>	libsocket	-l socket

Description:

The *ioctl()* function manipulates the underlying parameters of files. In particular, it can be used to control many of the operating attributes of files (such as the attributes of terminals).

The *ioctl_socket()* function is an optimized version of *ioctl()* that provides special handling for commands that use embedded pointers (see “[Commands with special](#)

handling in ioctl_socket() (p. 1206),” below). It uses *ioctl()* for commands that don't require special handling. The *fd* argument must be a socket for *ioctl_socket()*.



If you're using QNX Neutrino 6.4.1 or earlier, use *ioctl_socket()* instead of *ioctl()* in network applications such as packet filters.

In QNX Neutrino 6.5.0 and later, *ioctl()* handles embedded pointers, so you don't have to use *ioctl_socket()* instead.

The *request* argument determines whether the subsequent arguments are an “in” or “out” parameter; it also specifies the size of the arguments in bytes.

How *ioctl()* commands map to other commands

Some *ioctl()* commands map to calls to *fcntl()* (p. 623), *tcgetattr()* (p. 3347), *tcsetattr()* (p. 3364), and *tcsetsid()* (p. 3369). Other commands are transformed into a different *devctl()* (p. 403) command, and the rest are simply passed to *devctl()*. Here's a summary:

<i>ioctl()</i> command	Description	Maps to:
FIOASYNC	Set or clear asynchronous I/O	<i>fcntl()</i> : F_GETFL (which becomes a <i>devctl()</i> DCMD_ALL_GETFLAGS), followed by <i>fcntl()</i> : F_SETFL (which becomes a <i>devctl()</i> DCMD_ALL_SETFLAGS), setting or clearing O_ASYNC
FIOCLEX	Set “close on exec” on a file descriptor	<i>fcntl()</i> : F_SETFD with FD_CLOEXEC
FIOGETOWN	Get the owner	<i>fcntl()</i> : F_GETOWN
FIONBIO	Set or clear non-blocking I/O	<i>fcntl()</i> : F_GETFL (which becomes a <i>devctl()</i> DCMD_ALL_GETFLAGS), followed by <i>fcntl()</i> : F_SETFL (which becomes a <i>devctl()</i> DCMD_ALL_SETFLAGS), setting or clearing O_NONBLOCK
FIONCLEX	Remove “close on exec”	<i>fcntl()</i> : F_SETFD with ~FD_CLOEXEC

<i>ioctl()</i> command	Description	Maps to:
FIONREAD	Get the number of bytes to read	<i>devctl()</i> : DCMD_CHR_ISCHARS
FIOSETOWN	Set the owner	<i>fcntl()</i> : F_SETOWN
NOSIOCGIFCONF	Get ifnet list	Passed to <i>devctl()</i>
OSIOCGIFADDR	Get ifnet address	Passed to <i>devctl()</i>
OSIOCGIFBRDADDR	Get broadcast address	Passed to <i>devctl()</i>
OSIOCGIFCONF	Get ifnet list	Passed to <i>devctl()</i>
OSIOCGIFDSTADDR	Get point-to-point address	Passed to <i>devctl()</i>
OSIOCGIFNETMASK	Get net address mask	Passed to <i>devctl()</i>
SIOCADDMULTI	Add multicast address	Passed to <i>devctl()</i>
SIOCADDRT	Add route	Passed to <i>devctl()</i>
SIOCAIFADDR	Add or change an interface alias	Passed to <i>devctl()</i>
SIOCALIFADDR	Add interface address	Passed to <i>devctl()</i>
SIOCATMARK	At out-of-band mark?	Passed to <i>devctl()</i>
SIOCDELRARP	Delete ARP entry	Passed to <i>devctl()</i>
SIOCDELMULTI	Delete multicast address	Passed to <i>devctl()</i>
SIOCDELRT	Delete route	Passed to <i>devctl()</i>
SIOCDELIFADDR	Delete interface address	Passed to <i>devctl()</i>
SIOCDELPHYADDR	Delete gif addresses	Passed to <i>devctl()</i>
SIOCDELIFADDR	Delete interface address	Passed to <i>devctl()</i>
SIOCGARP	Get ARP entry	Passed to <i>devctl()</i>
SIOCGDRVSPEC	Get driver-specific parameters	Passed to <i>devctl()</i>
SIOCGETSGCNT	Get the source group packet count	Passed to <i>devctl()</i>
SIOCGETVIFCNT	Get the packet count for a virtual interface	Passed to <i>devctl()</i>
SIOCGHIWAT	Get high watermark	Passed to <i>devctl()</i>
SIOCGIFADDR	Get the interface address	Passed to <i>devctl()</i>

<i>ioctl()</i> command	Description	Maps to:
SIOCGIFALIAS	Get interface alias	Passed to <i>devctl()</i>
SIOCGIFASYNCMAP	Get ppp asyncmap	Passed to <i>devctl()</i>
SIOCGIFBRDADDR	Get broadcast address	Passed to <i>devctl()</i>
SIOCGIFCAP	Get capabilities	Passed to <i>devctl()</i>
SIOCGIFCONF	Return a list of interface (transport layer) addresses	Passed to <i>devctl()</i>
SIOCGIFDLT	Get data link type	Passed to <i>devctl()</i>
SIOCGIFDSTADDR	Get point-to-point address	Passed to <i>devctl()</i>
SIOCGIFFLAGS	Get ifnet flags	Passed to <i>devctl()</i>
SIOCGIFGENERIC	Generic interface get op	Passed to <i>devctl()</i>
SIOCGIFMEDIA	Get net media; <i>ioctl_socket()</i> provides special handling for this command.	Passed to <i>devctl()</i>
SIOCGIFMETRIC	Get interface metric	Passed to <i>devctl()</i>
SIOCGIFMTU	Get ifnet MTU	Passed to <i>devctl()</i>
SIOCGIFNETMASK	Get net address mask	Passed to <i>devctl()</i>
SIOCGIFPDSTADDR	Get gif pdst address	Passed to <i>devctl()</i>
SIOCGIFPSRCADDR	Get gif psrc address	Passed to <i>devctl()</i>
SIOCGLIFADDR	Get interface address	Passed to <i>devctl()</i>
SIOCGLIFPHYADDR	Get gif addresses	Passed to <i>devctl()</i>
SIOCGLOWAT	Get low watermark	Passed to <i>devctl()</i>
SIOCGPGRP	Get process group	<i>fcntl()</i> : F_GETOWN
SIOCIFCREATE	Create clone interface	Passed to <i>devctl()</i>
SIOCIFDESTROY	Destroy clone interface	Passed to <i>devctl()</i>
SIOCIFGCLONERS	Get cloners; <i>ioctl_socket()</i> provides special handling for this command.	Passed to <i>devctl()</i>
SIOCSARP	Set ARP entry	Passed to <i>devctl()</i>

<i>ioctl()</i> command	Description	Maps to:
SIOCSDRVSPEC	Set driver-specific parameters	Passed to <i>devctl()</i>
SIOCSHIWAT	Set high watermark	Passed to <i>devctl()</i>
SIOCSIFADDR	Set ifnet address	Passed to <i>devctl()</i>
SIOCSIFASYNCMAP	Set ppp asyncmap	Passed to <i>devctl()</i>
SIOCSIFBRDADDR	Set broadcast address	Passed to <i>devctl()</i>
SIOCSIFCAP	Set capabilities	Passed to <i>devctl()</i>
SIOCSIFDSTADDR	Set point-to-point address	Passed to <i>devctl()</i>
SIOCSIFFLAGS	Set ifnet flags	Passed to <i>devctl()</i>
SIOCSIFGENERIC	Generic interface set op	Passed to <i>devctl()</i>
SIOCSIFMEDIA	Set net media	Passed to <i>devctl()</i>
SIOCSIFMETRIC	Set interface metric	Passed to <i>devctl()</i>
SIOCSIFMTU	Set ifnet MTU	Passed to <i>devctl()</i>
SIOCSIFNETMASK	Set net address mask	Passed to <i>devctl()</i>
SIOCSIFPHYADDR	Set gif address	Passed to <i>devctl()</i>
SIOCSLIFPHYADDR	Set gif addresses	Passed to <i>devctl()</i>
SIOCSLOWAT	Set low watermark	Passed to <i>devctl()</i>
SIOCSPGRP	Set process group	<i>fcntl()</i> : F_SETOWN
TCFLSH	Flush buffers	Passed to <i>devctl()</i>
TCGETA	Get the terminal's properties in a termio structure	<i>tcgetattr()</i>
TCGETS	Get the terminal's properties in a termios structure	<i>tcgetattr()</i>
TCSBRK	Send a break for a period of time	<i>devctl()</i> : DCMD_CHR_SERCTL
TCSETA	Set the terminal's properties	<i>tcsetattr()</i>
TCSETAF	Set the terminal's properties, waiting until all	<i>tcsetattr()</i>

<i>ioctl()</i> command	Description	Maps to:
	currently written data has been transmitted, and discarding any received but unread data	
TCSETAW	Set the terminal's properties, waiting until all currently written data has been transmitted	<i>tcsetattr()</i>
TCSETS	Set the terminal's properties from a <code>termios</code> structure	<i>devctl()</i> : DCMD_CHR_TCSETATTR
TCSETSF	Drain the output, flush the input, and set	Passed to <i>devctl()</i>
TCSETSW	Drain output, set	Passed to <i>devctl()</i>
TCXONC	Perform a flow-control operation on a data stream	<i>devctl()</i> : DCMD_CHR_TCFLOW
TIOCCBRK	Clear the break bit	<i>tcsetattr()</i>
TIOCCDTR	Clear the Data Terminal Ready line	<i>devctl()</i> : DCMD_CHR_LINESTATUS, followed by DCMD_CHR_SERCTL
TIOCDRAIN	Wait until output has drained	<i>devctl()</i> : DCMD_CHR_TCDRAIN
TIOCEXCL	Set exclusive use of tty	<i>fcntl()</i> : F_SETLK
TIOCFLUSH	Flush buffers	<i>devctl()</i> : DCMD_CHR_TCFLUSH
TIOCGETA	Get the terminal's properties in a <code>termios</code> structure	<i>devctl()</i> : DCMD_CHR_TCGETATTR
TIOCGETC	Get special characters	<i>tcgetattr()</i>
TIOCGETP	Get the terminal's parameters in a <code>sgttyb</code> structure	<i>tcgetattr()</i>
TIOCGETPGRP	Get the process group of the tty (POSIX)	Passed to <i>devctl()</i>

<i>ioctl()</i> command	Description	Maps to:
TIOCGLTTC	Get local special characters	<i>tcgetattr()</i>
TIOCGPGRP	Get the process group of the tty	<i>devctl()</i> : DCMD_CHR_TCGETPGRP
TIOCGSIZE	Get window size	<i>devctl()</i> : DCMD_CHR_GETSIZE
TIOCGWINSZ	Get window size	<i>devctl()</i> : DCMD_CHR_GETSIZE
TIOCHPCL	Hang up on last close	<i>tcsetattr()</i>
TIOCLGET	Get local modes	<i>tcgetattr()</i>
TIOCLSET	Set entire local mode word	<i>tcsetattr()</i>
TIOCMBIC	Clear modem port B interrupt	<i>devctl()</i> : DCMD_CHR_LINESTATUS, followed by DCMD_CHR_SERCTL
TIOCMBIS	Set modem port B interrupt	<i>devctl()</i> : DCMD_CHR_LINESTATUS, followed by DCMD_CHR_SERCTL
TIOCMGET	Set the state of the modem lines	<i>devctl()</i> : DCMD_CHR_SERCTL
TIOCMSET	Set all modem bits	Passed to <i>devctl()</i>
TIOCNOTTY	Make this terminal not be the controlling terminal for the process	<i>tcsetsid()</i>
TIOCNXCL	Reset exclusive use of tty	<i>fcntl()</i> : F_SETLK
TIOCOUTQ	Output queue size	<i>devctl()</i> : DCMD_CHR_OSCHARS
TIOCPKT	Pty: set/clear packet mode	Passed to <i>devctl()</i>
TIOCSBRK	Set break bit	Passed to <i>devctl()</i>
TIOCSCTTY	Make the terminal become the controlling tty	<i>tcsetsid()</i>
TIOCSDTR	Set the Data Terminal Ready line	<i>devctl()</i> : DCMD_CHR_LINESTATUS,

<i>ioctl()</i> command	Description	Maps to:
		followed by DCMD_CHR_SERCTL
TIOCSETA	Set the terminal's properties from a <code>termios</code> structure	<i>devctl()</i> : DCMD_CHR_TCSETATTR
TIOCSETAF	Drain the output, flush the input, and set	<i>devctl()</i> : DCMD_CHR_TCSETATTRF
TIOCSETAW	Drain output, set	<i>devctl()</i> : DCMD_CHR_TCSETATTRD
TIOCSETC	Set special characters	<i>tcsetattr()</i>
TIOCSETN	Similar to <code>TIOCSETP</code> , but the changes are made immediately without discarding any data.	<i>tcsetattr()</i>
TIOCSETP	Set the terminal's parameters in a <code>sgttyb</code> structure. No change is made until all currently written data has been transmitted, at which point any received but unread data is discarded.	<i>tcsetattr()</i>
TIOCSETPGRP	Set the process group of the tty (POSIX)	Passed to <i>devctl()</i>
TIOCSINUSE	Set exclusive use of tty	<i>fcntl()</i> : <code>F_SETLK</code>
TIOCSLTC	Set local special characters	<i>tcsetattr()</i>
TIOCSGRP	Set the process group of the tty	<i>devctl()</i> : DCMD_CHR_TCSETPGRP
TIOCSSIZE	Set the window size	<i>devctl()</i> : DCMD_CHR_SETSIZE
TIOCSTART	Start output, like Ctrl-Q	Passed to <i>devctl()</i>
TIOCSTI	Simulate terminal input	<i>devctl()</i> : DCMD_CHR_TCINJECTC
TIOCSTOP	Stop output, like Ctrl-S	Passed to <i>devctl()</i>

<i>ioctl()</i> command	Description	Maps to:
TIOCSWINSZ	Set window size	<i>devctl()</i> : DCMD_CHR_SETSIZE
UIOCCMD	User control operator <i>n</i>	Passed to <i>devctl()</i>



If you're writing a resource manager, note that the [iofunc_devctl_default\(\)](#) (p. 1261) function handles the DCMD_ALL_GETFLAGS and DCMD_ALL_SETFLAGS commands; your resource manager will need to handle all other *devctl* commands that apply to it.

Commands with special handling in *ioctl_socket()*

The *ioctl_socket()* function provides special handling for the commands listed below. Unless otherwise noted, it eventually passes the command to [devctl\(\)](#) (p. 403).

These commands are for the Berkeley Packet Filter:

BIOCGDLTLIST

Get an array of the available types of the data link layer underlying the attached interface.

BIOCSETF

Set the BPF filter program.

These commands are for use with cryptography:

CIOCCRYPT

Request a symmetric-key (or unkeyed hash) operation.

CIOCGSESSION

Set up a session for symmetric crypto operations.

CIOCKEY

Perform an asymmetric crypto operation.

CRIOGET

Get a cloned file descriptor of `/dev/crypto` by calling [open\(\)](#) (p. 1905). This isn't necessary in QNX Neutrino, but we've provided it for compatibility and portability.

The following commands are for use with the Packet Filtering interface; for more information, see the entry for *pf* in the *Utilities Reference*:

- DIOCGGETSRCNODES

- DIOCGETSTATES
- DIOCIGETIFACES
- DIOCRADDADDRS
- DIOCRADDTABLES
- DIOCRCLRASTATS
- DIOCRCLRTSTATS
- DIOCRDELADDRS
- DIOCRDELTABLES
- DIOCRGETADDRS
- DIOCRGETASTATS
- DIOCRGETTABLES
- DIOCRGETTSTATS
- DIOCRINADEFINE
- DIOCRSETADDRS
- DIOCRSETTFLAGS
- DIOCRTSTADDRS
- DIOCXBEGIN
- DIOXC_COMMIT
- DIOCXROLLBACK

These commands are for use with sockets:

SIOCG80211

Get configuration or status information.

SIOCG80211NWID

Get the network ID.

SIOCG80211NWKEY

Get the values of the WEP keys.

SIOCG80211STATS, SIOCG80211ZSTATS

Get IEEE 80211 statistics.

SIOCGETVLAN

Get the vlan tag and parent for a given vlan interface.

SIOCGIFMEDIA

Get net media.

SIOCIFGCLOWERS

Get cloners.

SIOCS80211

Set configuration or status information.

SIOCS80211NWID

Set the network ID.

SIOCS80211NWKEY

Set the WEP key.

SIOCSETVLAN

Set the vlan tag and parent for a given vlan interface.

Returns:

A value based on the *request*, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid descriptor *fd*.

EINVAL

The *request* or optional variables aren't valid.

ENOBUFS

There wasn't enough memory available to allocate for data referred to by embedded pointers.

ENOTTY

The *fd* argument isn't associated with a character-special device; the specified *request* doesn't apply to the kind of object that the descriptor *fd* references.

Classification:

[POSIX 1003.1 XSR](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No

Safety:	
Thread	No

Caveats:

The *ioctl()* function is a Unix function that varies greatly from platform to platform.

iofdinfo()

Retrieve server attributes

Synopsis:

```
#include <sys/iomgr.h>

int iofdinfo( int filedes,
              unsigned flags,
              struct _fdinfo * info,
              char * path,
              int maxlen );
```

Arguments:

filedes

A file descriptor for the connection that you want to query.

flags

Specify `_FDINFO_FLAG_LOCALPATH` to return only the local path info (i.e. exclude the network path info).

info

NULL, or a pointer to an `_fdinfo` structure that contains the connection information defined in `<sys/iomgr.h>`. Specify NULL if it's not required.

path

A pointer to a buffer where the function can store the path associated with the file descriptor. Specify NULL if it's not required.

maxlen

The length of the buffer pointed to *path*.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofdinfo()* function retrieves the server's attribute information for the connection referred to by *filedes*.

Returns:

The length of the associated *filedes* pathname, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EFAULT**

A fault occurred in a server's address space when it tried to access the caller's message buffers.

EMSGSIZE

Insufficient space available in the server's buffer for the *fdinfo* data structure.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_acl()

Handle an IO_ACL message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_acl( resmgr_context_t *ctp,
               io_acl_t *msg,
               iofunc_ocb_t *ocb );
               iofunc_attr_t *attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to a [io_acl_t](#) (p. 1213) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_acl()* helper function implements handling of `_IO_ACL` (access control list) messages. It's called by the default handler, [iofunc_acl_default\(\)](#) (p. 1215).



In order to support ACLs, the *conf* member of the resource manager's `iofunc_mount_t` structure needs to have the `IOFUNC_PC_ACL` bit set.

For more information about ACLs, see Working with Access Control Lists (ACLs) in the QNX Neutrino *Programmer's Guide*.

io_acl_t structure

The `io_acl_t` structure holds the `_IO_ACL` message received by the resource manager:

```
struct _io_acl {
    _Uint16t                                type;
    _Uint16t                                combine_len;
    _Uint32t                                subtype;
    _Int32t                                 zero[2];
    /* struct _acl_header                    hdr; */
    /* void                                acl_data */
};

enum _io_acl_subtypes {
    _IO_ACL_GET,
    _IO_ACL_SET,
    _IO_ACL_CHK
};

struct _io_acl_reply {
    _Uint32t                                zero[4];
    /* struct _acl_header                    hdr; */
    /* void                                acl_data */
};

typedef union {
    struct _io_acl                          i;
    struct _io_acl_reply                     o;
} io_acl_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type `_io_acl` that contains the following members:

type

`_IO_ACL.`

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

subtype

One of the following:

- `_IO_ACL_GET` —
- `_IO_ACL_SET` —
- `_IO_ACL_CHK` —

nbytes

The number of bytes of data being passed with the command.

The commented-out declaration for *data* indicates that *nbytes* bytes of data immediately follow the `_io_acl` structure.

The *o* member of the `io_acl_t` message is a structure of type `_io_acl_reply` that currently contains no useful data. The commented-out declarations for *hdr* and *data* indicate that an `_acl_header` structure and additional data immediately follows the `_io_acl_reply` structure.

Returns:**EOK**

Success.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_acl_default()

Default handler for the acl callout

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_acl_default(
    resmgr_context_t *ctp,
    io_acl_t *msg,
    iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to a [io_acl_t](#) (p. 1213) structure that contains the message that the resource manager received; see the entry for *iofunc_acl()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_acl_default()* function provides default handling of `_IO_ACL` (access control lists) messages.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *acl* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all of the functions to their default values.

The `iofunc_acl_default()` function calls `iofunc_acl_track()` and [iofunc_acl\(\)](#) (p. 1212) to do the actual work.



In order to support ACLs, the `conf` member of the resource manager's `iofunc_mount_t` structure needs to have the `IOFUNC_PC_ACL` bit set.

For more information about ACLs, see Working with Access Control Lists (ACLs) in the QNX Neutrino *Programmer's Guide*.

Returns:

EOK

Success.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_attr_init()

Initialize the per-name attribute structure associated with a device

Synopsis:

```
#include <sys/iofunc.h>

void iofunc_attr_init ( iofunc_attr_t *attr,
                        mode_t mode,
                        iofunc_attr_t *dattr,
                        struct _client_info *info );
```

Arguments:

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that you want to initialize.

mode

The type and access permissions that you want to use for the resource. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

In general, there isn't a “correct” set of permissions to use; you should restrict them according to what you want other processes and users to be able to do with your resource manager.

dattr

NULL, or a pointer to a *iofunc_attr_t* structure that you want to use to initialize the structure pointed to by *attr*.

info

NULL, or a pointer to a *_client_info* structure that contains the information about a client connection. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

libc

Use the *-l c* option to *qcc* to link against this library. This library is usually included automatically.

Description:

The *iofunc_attr_init()* function initializes the per-name attributes that a resource manager associates with a device.

This function initializes the [iofunc_attr_t](#) (p. 1221) structure pointed to by *attr* with the information derived from the optional *dattr*, the *mode*, and the user and group IDs from the optional *info* client information structure.

The *count*, *rcount*, *wcount*, *rlocks* and *wlocks* counters are reset to zero.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_attr_lock()

Lock the attribute structure

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_attr_lock( iofunc_attr_t *attr );
```

Arguments:

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that you want to lock.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_attr_lock()* function locks the attribute structure that *attr* points to, preventing other threads in the resource manager from changing information.

Call this function (or [iofunc_attr_trylock\(\)](#) (p. 1226)) before you make any modifications to the attribute structure. After you're finished making modifications, call [iofunc_attr_unlock\(\)](#) (p. 1228) to release the lock.

Note that this is a *counting* locking mechanism. This means that a given thread can lock the attributes structure multiple times; it must then unlock the attributes structure a corresponding number of times in order to have the attributes structure considered unlocked. If another thread attempts to lock the structure while a thread has the structure locked, the other thread blocks.

Returns:

EOK

Success.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_attr_t

I/O attribute structure

Synopsis:

```
#include <sys/iofunc.h>

typedef struct _iofunc_attr {
    IOFUNC_MOUNT_T          *mount;
    uint32_t                flags;
    int32_t                 lock_tid;
    uint16_t                lock_count;
    uint16_t                count;
    uint16_t                rcount;
    uint16_t                wcount;
    uint16_t                rlocks;
    uint16_t                wlocks;
    struct _iofunc_mmap_list *mmap_list;
    struct _iofunc_lock_list *lock_list;
    void                    *lockobj;
    void                    *acl;
#if !defined(_IOFUNC_OFFSET_BITS) || _IOFUNC_OFFSET_BITS == 64

    #if __OFF_BITS__ == 64
        off_t                nbytes;
        ino_t                inode;
    #else
        off64_t              nbytes;
        ino64_t              inode;
    #endif
#elif _IOFUNC_OFFSET_BITS - 0 == 32
    #if __OFF_BITS__ == 32
        #if defined(__LITTLEENDIAN__)
            off_t            nbytes;
            off_t            nbytes_hi;
            ino_t            inode;
            ino_t            inode_hi;
        #elif defined(__BIGENDIAN__)
            off_t            nbytes_hi;
            off_t            nbytes;
            ino_t            inode_hi;
            ino_t            inode;
        #else
            #error endian not configured for system
        #endif
    #else
        #if defined(__LITTLEENDIAN__)
            int32_t          nbytes;
            int32_t          nbytes_hi;
            int32_t          inode;
            int32_t          inode_hi;
        #elif defined(__BIGENDIAN__)
            int32_t          nbytes_hi;
            int32_t          nbytes;
            int32_t          inode_hi;
            int32_t          inode;
        #else
            #error endian not configured for system
        #endif
    #endif
#endif
};
```

```

        #error endian not configured for system
    #endif
#endif
#else
    #error _IOFUNC_OFFSET_BITS value is unsupported
#endif
    uid_t          uid;
    gid_t          gid;
    time_t         mtime;
    time_t         atime;
    time_t         ctime;
    mode_t         mode;
    nlink_t        nlink;
    dev_t          rdev;
} iofunc_attr_t;

```

Description:

The `iofunc_attr_t` structure describes the attributes of the device that's associated with a resource manager. You typically use [iofunc_attr_init\(\)](#) (p. 1217) to initialize this structure.

The members include the following:

mount

A pointer a structure information about the mountpoint. By default, this structure is of type `iofunc_mount_t`, but you can specify your own structure by changing the `IOFUNC_MOUNT_T` manifest.

flags

Flags that your resource manager can set to indicate the state of the device. This member is a combination of the following flags:

IOFUNC_ATTR_ETIME

The access time is no longer valid. Typically set on a read from the resource.

IOFUNC_ATTR_CTIME

The change of status time is no longer valid. Typically set on a file info change.

IOFUNC_ATTR_DIRTY_ACL

The access control list has changed.

IOFUNC_ATTR_DIRTY_MODE

The mode has changed.

IOFUNC_ATTR_DIRTY_NLINK

The number of links has changed.

IOFUNC_ATTR_DIRTY_OWNER

The uid or the gid has changed.

IOFUNC_ATTR_DIRTY_RDEV

The *rdev* member has changed, e.g. *mknod()*.

IOFUNC_ATTR_DIRTY_SIZE

The size has changed.

IOFUNC_ATTR_DIRTY_TIME

One or more of *mtime*, *atime*, or *ctime* has changed.

IOFUNC_ATTR_MTIME

The modification time is no longer valid. Typically set on a write to the resource.

In addition to the above, your resource manager can use in any way the bits in the range defined by **IOFUNC_ATTR_PRIVATE** (see `<sys/iofunc.h>`).

lock_tid

The ID of the thread that has locked the attributes. To support multiple threads in your resource manager, you'll need to lock the attribute structure so that only one thread at a time is allowed to change it.

The resource manager layer automatically locks the attribute (using [iofunc_attr_lock\(\)](#) (p. 1219)) for you when certain handler functions are called (i.e. **IO_***).

lock_count

The number of times the thread has locked the attribute structure. You can lock the attributes by calling [iofunc_attr_lock\(\)](#) (p. 1219) or [iofunc_attr_trylock\(\)](#) (p. 1226); unlock them by calling [iofunc_attr_unlock\(\)](#) (p. 1228)



A thread must unlock the attributes as many times as it locked them.

count

The number of OCBs using this attribute in any manner. When this count is zero, no one is using this attribute.

rcount

The number of OCBs using this attribute for reading.

wcount

The number of OCBs using this attribute for writing.

rlocks

The number of read locks currently registered on the attribute.

wlocks

The number of write locks currently registered on the attribute.

mmap_list* and *lock_list

To manage their particular functionality on the resource, the *mmap_list* member is used by [iofunc_mmap\(\)](#) (p. 1294) and [iofunc_mmap_default\(\)](#) (p. 1298); the *lock_list* member is used by [iofunc_lock_default\(\)](#) (p. 1281). Generally, you shouldn't need to modify or examine these members.

lockobj

Attribute locking object.

acl

Access control lists.

nbytes

The number of bytes in the resource; your resource manager can change this value.

For a file, this would contain the file's size. For special devices (e.g. `/dev/null`) that don't support [lseek\(\)](#) (p. 1555) or have a radically different interpretation for [lseek\(\)](#), this field isn't used (because you wouldn't use any of the helper functions, but would supply your own instead.) In these cases, we recommend that you set this field to zero, unless there's a meaningful interpretation that you care to put to it.

inode

This is a mountpoint-specific inode that must be unique per mountpoint. You can specify your own value, or 0 to have the Process manager fill it in for you. For filesystem type of applications, this may correspond to some on-disk structure. In any case, the interpretation of this field is up to you.

uid* and *gid

The user ID and group ID of the owner of this resource. These fields are updated automatically by the *chown()* helper functions (e.g. *iofunc_chown_default()*) and are referenced in conjunction with the *mode* member for access-granting purposes by the *open()* help functions (e.g. *iofunc_open_default()*).

mtime, atime, and ctime

POSIX time members:

- *mtime* — modification time (*write()* updates this).
- *atime* — access time (*read()* updates this).
- *ctime* — change of status time (*write()*, *chmod()* and *chown()* update this).



One or more of the three time members may be *invalidated* as a result of calling an iofunc-layer function. To see if a time member is invalid, check the *flags* member. This is to avoid having each and every I/O message handler go to the kernel and request the current time of day, just to fill in the attribute structure's time member(s).

To fill the members with the correct time, call [iofunc_time_update\(\)](#) (p. 1364).

mode

The resource's mode (e.g. type, permissions). Valid modes may be selected from the *S_** series of constants in `<sys/stat.h>`; see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

nlink

The number of links to this particular name; your resource manager can modify this member. For names that represent a directory, this value must be at least 2 (one for the directory itself, one for the `./` entry in it).

rdev

The device number for a character special device and the *rdev* number for a named special device.

Classification:

[QNX Neutrino](#) (p. 46)

iofunc_attr_trylock()

Try to lock the attribute structure

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_attr_trylock( iofunc_attr_t *attr );
```

Arguments:

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that you want to lock.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_attr_trylock()* function attempts to lock the attribute structure *attr*, preventing other threads in the resource manager from changing information. If it can't lock *attr* immediately, it returns `EBUSY`.

Call this function (or [iofunc_attr_lock\(\)](#) (p. 1219)) before you make any modifications to the attribute structure. After you're finished making modifications, call [iofunc_attr_unlock\(\)](#) (p. 1228) to release the lock.

Note that this is a *counting* locking mechanism. This means that a given thread can lock the attributes structure multiple times; it must then unlock the attributes structure a corresponding number of times in order to have the attributes structure considered unlocked. If another thread attempts to lock the structure while a thread has the structure locked, the other thread will block.

Returns:

EOK

Success.

EBUSY

The calling thread couldn't lock the attributes immediately.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_attr_unlock()

Unlock the attribute structure

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_attr_unlock( iofunc_attr_t *attr );
```

Arguments:

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that you want to unlock.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_attr_unlock()* function unlocks the attribute structure *attr*, allowing other threads in the resource manager to change information.

Use this function in conjunction with [iofunc_attr_lock\(\)](#) (p. 1219) or [iofunc_attr_trylock\(\)](#) (p. 1226); call *iofunc_attr_unlock()* after you've made modifications to the attribute structure. You must unlock the structure as many times as you locked it.

Returns:

EOK

Success.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_check_access()

Check access permissions

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_check_access(
    resmgr_context_t *ctp,
    const iofunc_attr_t *attr,
    mode_t checkmode,
    const struct _client_info *info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that defines the characteristics of the device that's associated with the resource manager.

checkmode

The type and access permissions that you want to check for the resource. For more information, see below.

info

A pointer to a `_client_info` structure that contains the information about a client connection. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356). You can get this structure by calling [iofunc_client_info_ext\(\)](#) (p. 1246).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_check_access()* function verifies that the client is allowed access to the resource, as specified by a combination of who the client is (*info*), and the resource attributes *attr->mode*, *attr->uid* and *attr->gid*. Access is tested based upon the *checkmode* parameter.

The *checkmode* parameter determines which checks are done. It's a bitwise OR of the following constants:

S_ISUID

Verifies that the effective user ID of the client is equal to the user ID specified by the *attr->uid* member.

S_ISGID

Verifies that the effective group ID or one of the supplementary group IDs of the client is equal to the group ID specified by the *attr->gid* member.

S_IREAD

Verifies that the client has READ access to the resource as specified by *attr->mode*.

If the client's effective user ID matches that of *attr->uid*, then the permission check is made against the owner permission field of *attr->mode* (mask 0700 octal).

If the client's effective user ID doesn't match that of *attr->uid*, then if the client's effective group ID matches that of *attr->gid*, or one of the client's supplementary group IDs matches *attr->gid*, the check is made against the group permission field of *attr->mode* (mask 0070 octal).

If none of the group fields match, the check is made against the other permission field of *attr->mode* (mask 0007 octal).

S_IWRITE

Same as S_IREAD, except WRITE access is tested.

S_IEXEC

Same as S_IREAD, except EXECUTE access is tested. Note that since most resource managers don't actually execute code, the execute access is typically used in its directory sense, i.e. to test for directory accessibility, rather than execute access.

The S_ISUID and S_ISGID flags are mutually exclusive, that is, you may specify at most one of them. In conjunction with the S_ISUID and S_ISGID flags, you may

specify zero or more of the S_IREAD, S_IWRITE, and S_IEXEC flags. If no flags are specified, the permission checks are performed for privileged (root) access.

Here's some pseudo-code to try to explain this:

```
if superuser:
    return EOK

if S_ISUID and effective user ID == file user ID:
    return EOK

if S_ISGID and effective group ID == file group ID:
    return EOK

if S_IREAD or S_IWRITE or S_IEXEC:
    if caller's user ID == effective user ID:
        if all permissions are set in file's owner mode bits:
            return EOK
        else:
            return EACCES

    if ( caller's group ID or supplementary group IDs ) ==
        effective group ID:
        if all permissions are set in file's group mode bits:
            return EOK
        else:
            return EACCES

    if all permissions are set in file's other mode bits:
        return EOK
    else:
        return EACCES

return EPERM
```

Returns:

EACCES

The client doesn't have permissions to do the operation.

ENOSYS

NULL was passed for *info* structure.

EOK

Successful completion.

EPERM

The group ID or owner ID didn't match.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

iofunc_chmod()

Handle an `_IO_CHMOD` message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_chmod ( resmgr_context_t *ctp,
                  io_chmod_t *msg,
                  iofunc_ocb_t *ocb,
                  iofunc_attr_t *attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_chmod_t](#) (p. 1235) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_chmod()* helper function implements POSIX semantics for the client's *chmod()* call, which is received as an `_IO_CHMOD` message by the resource manager.

The *iofunc_chmod()* function verifies that the client has the necessary permissions to effect a *chmod()* on the attribute. If so, the *chmod()* is performed, modifying elements of the *ocb->attr* structure. This function takes care of updating the *IOFUNC_ATTR_CTIME*, *IOFUNC_ATTR_DIRTY_TIME*, and *IOFUNC_ATTR_DIRTY_MODE* bits in *ocb->attr->flags*. You can use [iofunc_time_update\(\)](#) (p. 1364), to update the appropriate time fields in *ocb->attr*.

You can use *iofunc_chmod()*, for example, in a filesystem manager, where an *_IO_CHMOD* message was received, and the filesystem code must now write the values to the medium. The filesystem code may wish to block the client thread until the data was actually written to the medium. Contrast this scenario to the behavior of [iofunc_chmod_default\(\)](#) (p. 1237), which calls this routine, and replies to the client thread.

io_chmod_t structure

The *io_chmod_t* structure holds the *_IO_CHMOD* message received by the resource manager:

```
struct _io_chmod {
    uint16_t          type;
    uint16_t          combine_len;
    mode_t            mode;
};

typedef union {
    struct _io_chmod      i;
} io_chmod_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type *_io_chmod* that contains the following members:

type

_IO_CHMOD.

combine_len

If the message is a combine message, *_IO_COMBINE_FLAG* is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

mode

The new mode. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Returns:

EOK

Successful completion.

EROFS

An attempt was made to chmod on a read-only filesystem.

EACCES

The client doesn't have permissions to do the operation.

EPERM

The group ID or owner ID didn't match.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_chmod_default()

Default handler for _IO_CHMOD messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_chmod_default( resmgr_context_t *ctp,
                          io_chmod_t *msg,
                          iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_chmod_t](#) (p. 1235) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_chmod()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_chmod_default()* function implements POSIX semantics for the client's *chmod()* call, which is received as an `_IO_CHMOD` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *chmod* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all the functions to their default values.

The `iofunc_chmod_default()` function calls [iofunc_chmod\(\)](#) (p. 1234) to do the actual work.

Returns:**EOK**

Successful completion.

EROFS

An attempt was made to `chmod()` on a read-only filesystem.

EACCES

The client doesn't have permissions to do the operation.

EPERM

The group ID or owner ID didn't match.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_chown()

Handle an `_IO_CHOWN` message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_chown ( resmgr_context_t *ctp,
                  io_chown_t *msg,
                  iofunc_ocb_t *ocb,
                  iofunc_attr_t *attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_chown_t](#) (p. 1240) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_chown()* helper function implements POSIX semantics for the client's *chown()* call, which is received as an `_IO_CHOWN` message by the resource manager.

The *iofunc_chown()* function verifies that the client has the necessary permissions to effect a chown on the attribute. If so, the chown is performed, modifying elements of

the `ocb->attr` structure. As per POSIX 1003.1, if the client isn't root, `iofunc_chown()` clears the set-user-id and set-group-id bits in the `ocb->attr->mode` member.

This function takes care of updating the `IOFUNC_ATTR_CTIME`, `IOFUNC_ATTR_DIRTY_TIME`, and `IOFUNC_ATTR_DIRTY_MODE` bits in `ocb->attr->flags`. You can use `iofunc_time_update()` (p. 1364), to update the appropriate time fields in `ocb->attr`.

io_chown_t structure

The `io_chown_t` structure holds the `_IO_CHOWN` message received by the resource manager:

```
struct _io_chown {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       gid;
    int32_t       uid;
};

typedef union {
    struct _io_chown i;
} io_chown_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_chown` that contains the following members:

type

`_IO_CHOWN`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

gid

The new group ID.

uid

The new user ID.

Returns:

EOK

Successful completion.

EROFS

An attempt was made to chown on a read-only filesystem.

EACCES

The client doesn't have permissions to do the operation.

EPERM

The group ID or owner ID didn't match.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_chown_default()

Default handler for _IO_CHOWN messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_chown_default( resmgr_context_t *ctp,
                        io_chown_t *msg,
                        iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_chown_t](#) (p. 1240) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_chown()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_chown_default()* function implements POSIX semantics for the client's *chown()* call, which is received as an `_IO_CHOWN` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *chown* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all of the functions to their default values.

The `iofunc_chown_default()` function calls `iofunc_chown()` (p. 1239) to do the actual work.

Returns:**EOK**

Successful completion.

EROFS

An attempt was made to chown on a read-only filesystem.

EACCES

The client doesn't have permissions to do the operation.

EPERM

The group ID or owner ID didn't match.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_client_info_able()

Return information about a client connection, including abilities

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_client_info_able ( resmgr_context_t * ctp,
                             const int ioflag,
                             struct _client_info ** info,
                             int flags,
                             struct _client_able abilities[],
                             const int nable );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ioflag

Zero, or the constant `O_REALIDS`. This argument is passed in the `_IO_OPEN` message during an open request. If you specify `O_REALIDS`, *iofunc_client_info_able()* swaps the real and effective values of the user and group IDs before returning. This is a QNX Neutrino extension, to swap real and effective user and group IDs in an atomic operation.

info

The address of a location where the function can store a pointer to a [_client_info](#) (p. 364) structure that the function can fill with information about the client. For information about this structure, see [ConnectClientInfoExt\(\)](#) (p. 363).

flags

Zero or more of the following:

- `_NTO_CLIENTINFO_GETGROUPS` — get the supplementary group IDs

able

An array of `struct _client_able` structures that specifies the abilities that the client must have; see [ConnectClientInfoAble\(\)](#) (p. 360).

nable

The number of entries in the *able* array.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *iofunc_client_info_able()* function fetches the *info* structure for the client, including its abilities. It calls [ConnectClientInfoAble\(\)](#) (p. 360) to gather the information, based on the server connection ID found in *ctp->info.scoid*.

Call [iofunc_client_info_ext_free\(\)](#) (p. 1248) to free the *_client_info* structure when you're done with it.

Returns:**EFAULT**

A fault occurred when the kernel tried to access the *info* buffer provided.

EINVAL

The client process is no longer valid.

EOK

Successful completion.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_client_info_ext()

Return information about a client connection

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_client_info_ext ( resmgr_context_t * ctp,
                           const int ioflag,
                           struct _client_info ** info,
                           int flags );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ioflag

Zero, or the constant `O_REALIDS`. This argument is passed in the `_IO_OPEN` message during an open request. If you specify `O_REALIDS`, *iofunc_client_info_ext()* swaps the real and effective values of the user and group IDs before returning. This is a QNX Neutrino extension, to swap real and effective user and group IDs in an atomic operation.

info

The address of a location where the function can store a pointer to a [_client_info](#) (p. 364) structure that the function can fill with information about the client. For information about this structure, see [ConnectClientInfoExt\(\)](#) (p. 363).

flags

Zero or more of the following:

- `_NTO_CLIENTINFO_GETGROUPS` — get the supplementary group IDs

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_client_info_ext()* function fetches the *info* structure for the client. It calls [ConnectClientInfoExt\(\)](#) (p. 363) to gather the information, based on the server connection ID found in *ctp->info.scoId*.

Call [iofunc_client_info_ext_free\(\)](#) (p. 1248) to free the *_client_info* structure when you're done with it.

Returns:**EFAULT**

A fault occurred when the kernel tried to access the *info* buffer provided.

EINVAL

The client process is no longer valid.

EOK

Successful completion.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_client_info_ext_free()

Free the structure used for information about a client connection

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_client_info_ext_free ( struct _client_info ** info
    );
```

Arguments:

info

The address of a pointer to the `_client_info` structure that you want to free.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_client_info_ext_free()* function frees the memory allocated by a call to [iofunc_client_info_ext\(\)](#) (p. 1246).

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_close_dup()

Free all locks allocated for the client process

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_close_dup( resmgr_context_t* ctp,
                     io_close_t* msg,
                     iofunc_ocb_t* ocb,
                     iofunc_attr_t* attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_close_t](#) (p. 1249) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_close_dup()* helper function handles a `_IO_CLOSE` message. This function frees all locks allocated for the client process on the file descriptor and performs any POSIX-related cleanup required when a duplicated *ocb* is detached.

io_close_t structure

The `io_close_t` structure holds the `_IO_CLOSE` message received by the resource manager:

```

struct _io_close {
    uint16_t          type;
    uint16_t          combine_len;
};

typedef union {
    struct _io_close    i;
} io_close_t;
    
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_close` that contains the following members:

type

`_IO_CLOSE`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see the Combine Messages chapter of *Writing a Resource Manager*.

Returns:

EOK

Success.

Anything else

An error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_close_dup_default()

Default handler for _IO_CLOSE messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_close_dup_default(
    resmgr_context_t *ctp,
    io_close_t *msg,
    iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_close_t](#) (p. 1249) structure that contains the message that the resource manager received. For more information, see the documentation for [iofunc_close_dup\(\)](#).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [iofunc_close_dup_default\(\)](#) function implements default actions for the `_IO_CLOSE` message. This function simply calls [iofunc_close_dup\(\)](#) (p. 1249) to do the actual work.

You can place [iofunc_close_dup_default\(\)](#) directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the `io_funcs` argument to [resmgr_attach\(\)](#) (p. 2580), at the `close_dup` position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.



If your resource manager uses [iofunc_lock_default\(\)](#) (p. 1281), you *must* use both [iofunc_close_dup_default\(\)](#) and [iofunc_unblock_default\(\)](#) (p. 1368) in their respective handlers, as they provide necessary ancillary functionality for managing file locks. This is because file locks are owned by the process, and aren't inherited by the child process.

Returns:

EOK

Successful completion.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_close_ocb()

Return the memory allocated for an OCB

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_close_ocb( resmgr_context_t* ctp,
                     iofunc_ocb_t* ocb,
                     iofunc_attr_t* attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_close_ocb()* function detaches the OCB specified by *ocb*, and releases the memory associated with it.



This function assumes that *ocb* points to an `iofunc_ocb_t`. If you encapsulate `iofunc_ocb_t` in your own OCB it must be the first field of your OCB; otherwise, you can't call this function. If you provide an *ocb_free()* function in the mount structure then it's called at this point. This means that at least the `iofunc_ocb_t` portion of your OCB is no longer valid after *iofunc_close_ocb()* returns.

The `iofunc_close_ocb()` function calls [iofunc_ocb_detach\(\)](#) (p. 1315) on your behalf.

Returns:

EOK

Success.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_close_ocb_default()

Return the memory allocated for an OCB

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_close_ocb_default( resmgr_context_t* ctp,
                             void* reserved,
                             iofunc_ocb_t* ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

reserved

This argument must be passed as NULL.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_close_ocb_default()* function detaches the OCB specified by *ocb*, and releases the memory associated with it.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *close_ocb* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all of the functions to their default values.



This function assumes that *ocb* points to an *iofunc_ocb_t*. If you encapsulate *iofunc_ocb_t* in your own OCB, it must be the first field of your OCB;

otherwise, you can't call this function. If you provide an *ocb_free()* function in the mount structure, it's called at this point. This means that at least the *iofunc_ocb_t* portion of your OCB is no longer valid after *iofunc_close_ocb()* returns.

The *iofunc_close_ocb_default()* function calls [iofunc_close_ocb\(\)](#) (p. 1253) to do the actual work.

Returns:**EOK**

Success.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_devctl()

Handle an `_IO_DEVCTL` message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_devctl( resmgr_context_t *ctp,
                  io_devctl_t *msg,
                  iofunc_ocb_t *ocb,
                  iofunc_attr_t *attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_devctl_t](#) (p. 1258) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_devctl()* helper function implements POSIX semantics for the client's *devctl()* call, which is received as an `_IO_DEVCTL` message by the resource manager. This function handles the `DCMD_ALL*` functionality.

This function handles at least the following device control messages:

DCMD_ALL_GETFLAGS

Implements the functionality of the *fcntl()* (p. 623) get-flags command.

DCMD_ALL_SETFLAGS

Implements the functionality of the *fcntl()* set-flags command.

DCMD_ALL_GETMOUNTFLAGS

Returns the mount flag (*mount->flags*) for a resource that has a mount structure defined, else returns a mount flag of zero.

The supported mount flags (bitmask values) for DCMD_ALL_GETMOUNTFLAGS include:

_MOUNT_READONLY

Read only.

_MOUNT_NOEXEC

Can't exec from filesystem.

_MOUNT_NOSUID

Don't honor setuid bits on filesystem.

Any other device control messages return ENOTTY.

io_devctl_t structure

The *io_devctl_t* structure holds the *_IO_DEVCTL* message received by the resource manager:

```
struct _io_devctl {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       dcmd;
    int32_t       nbytes;
    int32_t       zero;
    /* char       data[nbytes]; */
};

struct _io_devctl_reply {
    uint32_t       zero;
    int32_t        ret_val;
    int32_t        nbytes;
    int32_t        zero2;
    /* char       data[nbytes]; */
};

typedef union {
    struct _io_devctl      i;
    struct _io_devctl_reply o;
} io_devctl_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type *_io_devctl* that contains the following members:

type`_IO_DEVCTL.`**combine_len**

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

dcmd

The device-control command to execute.

nbytes

The number of bytes of data being passed with the command.

The commented-out declaration for *data* indicates that *nbytes* bytes of data immediately follow the `_io_devctl` structure.

The `_DEVCTL_DATA()` macro gets a pointer to the data that follows the message. Call it like this:

```
data = _DEVCTL_DATA (msg->i);
```

The *o* member of the `io_devctl_t` message is a structure of type `_io_devctl_reply` that contains the following members:

ret_val

The value returned by the command.

nbytes

The number of bytes of data being returned.

The commented-out declaration for *data* indicates that *nbytes* bytes of data immediately follow the `_io_devctl_reply` structure.

Returns:**EOK**

Successful completion.

EINVAL

An attempt to set the flags for a resource that is synchronized, with no mount structure defined, or no synchronized I/O defined.

ENOTTY

An unsupported device control message was decoded.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_devctl_default()

Default handler for `_IO_DEVCTL` messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_devctl_default( resmgr_context_t *ctp,
                          io_devctl_t *msg,
                          iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_devctl_t](#) (p. 1258) structure that contains the message that the resource manager received. For more information, see the documentation for [iofunc_devctl\(\)](#).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [iofunc_devctl_default\(\)](#) function implements the default actions for `_IO_DEVCTL` messages.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *devctl* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The `iofunc_devctl_default()` function calls `iofunc_devctl()` (p. 1257) to do the actual work. If you write your own handler, you can use `iofunc_devctl_verify()` (p. 1263) to do some validation of the message.

Returns:**EOK**

Successful completion.

EINVAL

An attempt to set the flags for a resource that is synchronized, with no mount structure defined, or no synchronized I/O defined.

ENOTTY

An unsupported device control message was decoded.

_RESMGR_DEFAULT

An supported device control message that isn't a known `DCMD_ALL_*` command was decoded.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_devctl_verify()

Perform validation that's common to all handlers for _IO_DEVCTL messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_devctl_verify( const resmgr_context_t *const ctp,
                        const io_devctl_t      *const msg,
                        const iofunc_ocb_t      *const ocb,
                        const unsigned
                        requested_checks );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_devctl_t](#) (p. 1258) structure that contains the message that the resource manager received. For more information, see the documentation for [iofunc_devctl\(\)](#) (p. 1257).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

requested_checks

Bits that specify which checks to perform; the flags are defined in `iomsg.h` and include the following:

- `_IO_DEVCTL_VERIFY_ACC_IEXEC` — requires user-level exec permission. For more information, see the description of [iofunc_check_access\(\)](#) (p. 1230).
- `_IO_DEVCTL_VERIFY_ACC_IREAD` — requires user-level read permission.
- `_IO_DEVCTL_VERIFY_ACC_ISGID` — the message sender must have group permission to the device.
- `_IO_DEVCTL_VERIFY_ACC_ISUID` — the message sender must be owner of the device.

- `_IO_DEVCTL_VERIFY_ACC_IWRITE` — requires user-level write permission.
- `_IO_DEVCTL_VERIFY_DIR` — Ensure that the direction (read/write) of the IOCTL command matches the open mode flags in the OCB. Commands using `__DIOF` require read access, `__DIOT` requires write access, `__DIOTF` requires read and write access. Commands that use `__DION` don't require read or write access.
- `_IO_DEVCTL_VERIFY_LEN` — ensure that the size encoded in the IOCTL command matches what's encoded in the message.
- `_IO_DEVCTL_VERIFY_MSG_LEN` — check that the incoming message has enough space to hold the size of the data as indicated in the *devctl()* encoding.
- `_IO_DEVCTL_VERIFY_OCB_READ` — explicitly check that the OCB was opened with read access.
- `_IO_DEVCTL_VERIFY_OCB_WRITE` — explicitly check that the OCB was opened with write access.
- `_IO_DEVCTL_VERIFY_PRIV` — ensure that the caller has `root` access.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_devctl_verify()* function performs validation that's common to all resource manager *devctl()* handlers. Generally it's meant to be used to test the security and validity of incoming *devctl()* messages from a client, or the privileges of the client such as ensuring the client is `root`, the resource owner, and so on.



- The default handler doesn't call this function.
- This function returns on the first error that it finds. The order in which errors are checked is subject to change.

The most common use is ensuring that the client sending the message has actually opened the resource for read and/or write. This is usually done using the `_IO_DEVCTL_VERIFY_DIR` flag, which reads the direction of the *devctl()* command and ensures that there are matching `ocb->oflags`.

Another important flag is `_IO_DEVCTL_VERIFY_MSG_LEN`, which checks that the incoming message has enough space to hold the size of the data as indicated in the `devctl()` encoding.

Returns:

EOK on success, or a POSIX error code on failure (see the entry for [errno](#) (p. 539)). This function doesn't set `errno`.

Examples:

Check the direction of a command:

```
if (EOK != (status = iofunc_devctl_verify(ctp, msg, ocb, _IO_DEVCTL_VERIFY_
{
    return status;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_fdinfo()

Handle an `_IO_FDINFO` message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_fdinfo( resmgr_context_t * ctp,
                  iofunc_ocb_t * ocb,
                  iofunc_attr_t * attr,
                  struct _fdinfo * info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

NULL, or a pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

info

A pointer to a `_fdinfo` structure that the function fills with the information. This structure is defined in `<sys/iomgr.h>` as:

```
struct _fdinfo {
    uint32_t    mode; /* File mode */
    uint32_t    ioflag; /* Current io flags */
    uint64_t    offset; /* Current seek position */
    uint64_t    size; /* Current size of file */
    uint32_t    flags; /* _FDINFO_ */
    uint16_t    sflag; /* Share flags */
    uint16_t    count; /* File use count */
    uint16_t    rcount; /* File reader count */
    uint16_t    wcount; /* File writer count */
    uint16_t    rlocks; /* Number of read locks */
    uint16_t    wlocks; /* Number of write locks */
    uint32_t    zero[6];
};
```

The `_fdinfo` structure is included in the reply part of a [io_fdinfo_t](#) (p. 1268) structure; for more information, see the documentation for [iofunc_fdinfo_default\(\)](#).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_fdinfo()` helper function provides the implementation for the client's [iofdinfo\(\)](#) (p. 1210) call, which is received as an `_IO_FDINFO` message by the resource manager.

The `iofunc_fdinfo()` function transfers the appropriate fields from the `ocb` and `attr` structures to the `info` structure. If `attr` is `NULL`, then the `attr` information comes from the structure pointed to by `ocb->attr`.

Returns:

EOK

Successful completion.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_fdinfo_default()

Default handler for _IO_FDINFO messages

Synopsis:

```
#include <sys/iomgr.h>

int iofunc_fdinfo_default( resmgr_context_t * ctp,
                          io_fdinfo_t * msg,
                          iofunc_ocb_t * ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_fdinfo_t](#) (p. 1269) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_fdinfo_default()* function provides the default handler for the client's [iofdinfo\(\)](#) (p. 1210) call, which is received as an `_IO_FDINFO` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *fdinfo* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The `iofunc_fdinfo_default()` function calls `iofunc_fdinfo()` (p. 1266) and `resmgr_pathname()` (p. 2639) to do the actual work.

io_fdinfo_t structure

The `io_fdinfo_t` structure holds the `_IO_FDINFO` message received by the resource manager:

```
struct _io_fdinfo {
    uint16_t          type;
    uint16_t          combine_len;
    uint32_t          flags;
    int32_t           path_len;
    uint32_t          reserved;
};

struct _io_fdinfo_reply {
    uint32_t          zero[2];
    struct _fdinfo    info;
    /* char           path[path_len + 1]; */
};

typedef union {
    struct _io_fdinfo    i;
    struct _io_fdinfo_reply o;
} io_fdinfo_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type `_io_fdinfo` that contains the following members:

type

`_IO_FDINFO.`

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

flags

Specify `_FDINFO_FLAG_LOCALPATH` to return only the local path info (i.e. exclude the network path info).

path_len

The size of the path reply buffers that follow the reply.

The *o* member is a structure of type `_io_fdinfo_reply` that contains the following members:

info

A `_fdinfo` structure that's defined (in `<sys/iomgr.h>`) as:

```
struct _fdinfo {
```

```

uint32_t    mode;    /* File mode */
uint32_t    ioflag; /* Current io flags */
uint64_t    offset; /* Current seek position */
uint64_t    size;   /* Current size of file */
uint32_t    flags;  /* _FDINFO_ */
uint16_t    sflag;  /* Share flags */
uint16_t    count;  /* File use count */
uint16_t    rcount; /* File reader count */
uint16_t    wcount; /* File writer count */
uint16_t    rlocks; /* Number of read locks */
uint16_t    wlocks; /* Number of write locks */
uint32_t    zero[6];
};

```

The commented-out declaration for *path* indicates that *path_len* + 1 bytes of data immediately follow the `_io_fdinfo_reply` structure.

Returns:

-1

Success; the resource manager library should return a one-part IOV to the client.

EMSGSIZE

Insufficient space is available in the server's buffer to receive the entire message.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_func_init()

Initialize the default POSIX-layer function tables

Synopsis:

```
#include <sys/iofunc.h>

void iofunc_func_init(
    unsigned nconnect,
    resmgr_connect_funcs_t *connect,
    unsigned nio,
    resmgr_io_funcs_t *io );
```

Arguments:

nconnect

The number of entries in the *connect* table that you want to fill. Typically, you pass `_RESMGR_CONNECT_NFUNCS` for this argument.

connect

A pointer to a [resmgr_connect_funcs_t](#) (p. 2592) structure that you want to fill with the default connect functions.

nio

The number of entries in the *io* table that you want to fill. Typically, you pass `_RESMGR_IO_NFUNCS` for this argument.

io

A pointer to a [resmgr_io_funcs_t](#) (p. 2610) structure that you want to fill with the default I/O functions.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_func_init()* function initializes the passed *connect* and *io* structures with the POSIX-layer default functions. For information about the default functions, see [resmgr_connect_funcs_t](#) (p. 2592) and [resmgr_io_funcs_t](#) (p. 2610).

The *nconnect* and *nio* arguments indicate how many entries this function should fill. This is in place to support forward compatibility.

Examples:

Fill a connect and I/O function table with the POSIX-layer defaults:

```
#include <sys/iofunc.h>

static resmgr_connect_funcs_t my_connect_functions;
static resmgr_io_funcs_t my_io_functions;

int main (int argc, char **argv)
{
...
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &my_connect_functions,
                     _RESMGR_IO_NFUNCS, &my_io_functions);

/*
 * At this point, the defaults have been filled in.
 * You may now override some of the default functions with
 * functions that you have written:
 */

    my_io_functions.io_read = my_io_read;
...
}
```

The above example initializes your connect and I/O function structures (*my_connect_functions* and *my_io_functions*) with the POSIX-layer defaults. If you didn't override any of the functions, your resource manager would behave like */dev/null* — any data written to it would be discarded, and an attempt to read data from it would immediately return an EOF.

Since this isn't desirable in most cases, you'll often provide functionality for some functions, such as reading, writing, and device control to your device. In the example above, we've explicitly supplied our own handler for reading from the device, via a function called *my_io_read()*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_link()

Link two directories

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_link( resmgr_context_t* ctp,
                 io_link_t* msg,
                 iofunc_attr_t* attr,
                 iofunc_attr_t* dattr,
                 struct _client_info* info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_link_t](#) (p. 1274) structure that contains the message that the resource manager received; see below.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the resource.

dattr

NULL, or a pointer to the [iofunc_attr_t](#) structure that describes the characteristics of the parent directory.

info

NULL, or a pointer to a [_client_info](#) structure that contains information about the client. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_link()* helper function links directory *attr* to *dattr* for context *ctp*. It's similar to the *iofunc_open()* (p. 1321) function:

The *iofunc_link()* function checks to see if the client (described by the optional *info* structure) has access to open the resource (name passed in the *msg* structure). The *attr* structure describes the resource's attributes, and the optional *dattr* structure defines the attributes of the parent directory (i.e. if *dattr* isn't NULL, it implies that the resource identified by *attr* is being created within the directory specified by *dattr*).

You can pass the *info* argument as NULL, in which case *iofunc_link()* obtains the client information itself via a call to *iofunc_client_info_ext()* (p. 1246). It is, of course, more efficient to get the client info once, rather than calling this function with NULL every time.

If you pass NULL in *info*, the function returns information about a client's connection in *info*, and an error constant.

io_link_t structure

The *io_link_t* structure holds the *_IO_CONNECT* message received by the resource manager:

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_link_t;
```

This message structure is a union of an input message (coming to the resource manager), *_io_connect* (p. 1189), and two possible output or reply messages (going back to the client):

- *_io_connect_link_reply* (p. 1196) if the reply is redirecting the client to another resource

Or:

- *_io_connect_ftype_reply* (p. 1195) if the reply consists of a status and a file type.

The reply includes the following additional information:

```
struct _io_resmgr_link_extra {
    uint32_t      nd;
    int32_t       pid;
    int32_t       chid;
    uint32_t      handle;
    uint32_t      flags;
    uint32_t      file_type;
    uint32_t      reserved[2];
};

typedef union _io_link_extra {
    struct _msg_info      info;      /* EXTRA_LINK (from client) */
    void                  *ocb;      /* EXTRA_LINK (from resmgr functions) */
    char                  path[1];   /* EXTRA_SYMLINK */
    struct _io_resmgr_link_extra resmgr; /* EXTRA_RESMGR_LINK */
} io_link_extra_t;
```

info

A pointer to a [_msg_info](#) (p. 1731) structure.

Returns:**EOK**

Success.

EBADFSYS.

NULL was passed in *attr* and *dattr*.

EFAULT

A fault occurred when the kernel tried to access the *info* buffer.

EINVAL

The client process is no longer valid.

ENOSYS

NULL was passed in *info*.

EPERM

The group ID or owner ID didn't match.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_lock()

Lock a resource

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_lock( resmgr_context_t * ctp,
                 io_lock_t * msg,
                 iofunc_ocb_t * ocb,
                 iofunc_attr_t * attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the `io_lock_t` structure that contains the message that the resource manager received.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *iofunc_lock()* does what is required for POSIX locks. For more information, see [lockf\(\)](#) (p. 1535).

Returns:

EACCES or EAGAIN

The operation is F_TLOCK or F_TEST, and the section is already locked by another process.

EAGAIN

The operation is F_LOCK or F_TLOCK, and the file is mapped with [mmap\(\)](#) (p. 1656).

EBADF

The file descriptor isn't valid, or the operation is F_LOCK or F_TLOCK, and the file descriptor isn't open for writing.

EDEADLK

The operation is F_LOCK, and a deadlock was detected.

EINTR

A signal was caught during execution of the function.

EINVAL

The operation isn't one of F_LOCK, F_TLOCK, F_TEST, or F_ULOCK, or the size plus the current file offset is less than 0.

ENOMEM

The system can't allocate sufficient memory to store lock resources.

EOPNOTSUPP or EINVAL

The implementation doesn't support the locking of files of the type indicated by the file descriptor.

EOVERFLOW

The offset of the first, or if the size isn't 0 then the last, byte in the requested section can't be represented correctly in an object of type `off_t`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

iofunc_lock_alloc()

Allocate memory to lock structures

Synopsis:

```
#include <sys/iofunc.h>

iofunc_lock_list_t *iofunc_lock_alloc
( resmgr_context_t *ctp,
  IOFUNC_OCB_T *ocb,
  size_t size );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ocb

A pointer to the the Open Control Block (typically a [iofunc_ocb_t](#) (p. 1319) structure) that was created when the client opened the resource.

size

The amount of memory that you want to allocate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *iofunc_lock_alloc()* is used by [iofunc_lock\(\)](#) (p. 1276) to allocate memory to lock structures.



Because the *malloc()* implementation uses signed, 32-bit integers to represent the size internally, you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, *iofunc_lock_alloc()* indicates an error of `ENOMEM`.

Returns:

A pointer to a zeroed buffer that the POSIX layer uses for locks, or **NULL** if no memory could be allocated ([errno](#) (p. 539) is set).

Errors:**ENOMEM**

Not enough memory.

EOK

No error.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_lock_default()

Default handler for `_IO_LOCK` messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_lock_default( resmgr_context_t * ctp,
                        io_lock_t * msg,
                        iofunc_ocb_t * ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the `io_lock_t` structure that contains the message that the resource manager received; see [iofunc_lock\(\)](#) (p. 1276).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_lock_default()* function implements POSIX semantics for the `_IO_LOCK` message (generated as a result of a client [fcntl\(\)](#) (p. 623) call).

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *lock* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The *iofunc_lock_default()* function verifies that the client has the necessary permissions to effect a lock on the resource. This includes checking for read and write permissions

against the type of lock being effected. If so, the lock is performed, modifying elements of the `ocb->attr` structure, and updating `ocb->attr->locklist` to reflect the new lock. This function calls [iofunc_lock\(\)](#) (p. 1276) to do the actual work.



If your resource manager calls `iofunc_lock_default()`, it must call [iofunc_close_dup_default\(\)](#) (p. 1251) and [iofunc_unblock_default\(\)](#) (p. 1368) in their respective handlers.

Returns:

EOK

Successful completion.

EINVAL

An invalid range was specified for the lock operation, or an invalid lock operation was attempted.

EBADF

An attempt to perform a read lock on a write-only resource, or a write lock on a read-only resource was attempted.

ENOMEM

Insufficient memory exists to allocate an internal lock structure.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_lock_free()

Return memory allocated for lock structures

Synopsis:

```
#include <sys/iofunc.h>

void iofunc_lock_free( iofunc_lock_list_t* lock,
                      size_t size );
```

Arguments:

lock

A pointer to the `iofunc_lock_list_t` list that you want to free.

size

The amount of memory that you want to free.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `iofunc_lock_free()` frees *lock* structures allocated by [iofunc_lock_alloc\(\)](#) (p. 1279).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_lock_ocb_default()

Default handler for the lock_ocb callout

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_lock_ocb_default( resmgr_context_t *ctp,
                             void *reserved,
                             iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

reserved

This argument must be NULL.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_lock_ocb_default()* function calls [iofunc_attr_lock\(\)](#) (p. 1219) to enforce locking on the attributes for the group of messages that were sent by the client.

You can place this function directly into the *io_funcs* table passed to [resmgr_attach\(\)](#) (p. 2580), at the *lock_ocb* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

Returns:

EOK

Success.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_lseek()

Handle an `_IO_LSEEK` message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_lseek ( resmgr_context_t* ctp,
                  io_lseek_t* msg,
                  iofunc_ocb_t* ocb,
                  iofunc_attr_t* attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_lseek_t](#) (p. 1287) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_lseek()* helper function implements POSIX semantics for the client's [lseek\(\)](#) (p. 1555) call, which is received as an `_IO_LSEEK` message by the resource manager.

The *iofunc_lseek()* function handles the three different *whence* cases: `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`, updating the `ocb->offset` field with the new position.

Note that if the IOFUNC_MOUNT_32BIT flag isn't set in the mount structure, *iofunc_lseek()* handles 64-bit position offsets. If the flag is set (meaning this device supports only 32-bit offsets), the resulting offset value is treated as a 32-bit offset, and if it overflows 32 bits, it's truncated to LONG_MAX. Also, this function handles combine messages correctly, simplifying the work required to support lseek.

io_lseek_t structure

The *io_lseek_t* structure holds the _IO_LSEEK message received by the resource manager:

```
struct _io_lseek {
    uint16_t      type;
    uint16_t      combine_len;
    short         whence;
    uint16_t      zero;
    uint64_t      offset;
};

typedef union {
    struct _io_lseek      i;
    uint64_t              o;
} io_lseek_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type *_io_lseek* that contains the following members:

type

_IO_LSEEK.

combine_len

If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

whence

SEEK_SET, SEEK_CUR, or SEEK_END.

offset

The relative offset from the file position determined by the *whence* member.

The *o* member is the offset after the operation is complete.

Returns:

-1

Success; the resource manager library should return a one-part IOV to the client.

EOK

The message is a combine message, and no error occurred.

EINVAL

The *whence* member in the `_IO_LSEEK` message wasn't one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, or the resulting position after the offset was applied resulted in a negative number (overflow).

E_OVERFLOW

The resulting file offset is a value that can't be represented correctly in an object of type `off_t`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_lseek_default()

Default handler for `_IO_LSEEK` messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_lseek_default( resmgr_context_t* ctp,
                        io_lseek_t* msg,
                        iofunc_ocb_t* ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_lseek_t](#) (p. 1287) structure that contains the message that the resource manager received. For more information, see the documentation for [iofunc_lseek\(\)](#).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [iofunc_lseek_default\(\)](#) function implements POSIX semantics for the client's [lseek\(\)](#) (p. 1555) call, which is received as an `_IO_LSEEK` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *lseek* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The `iofunc_lseek_default()` function calls [iofunc_lseek\(\)](#) (p. 1286) to do the actual work.

Returns:

-1

Success; the resource manager library should return a one-part IOV to the client.

EOK

The message is a combine message, and no error occurred.

EINVAL

The *whence* member in the `_IO_LSEEK` message wasn't one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, or the resulting position after the offset was applied resulted in a negative number (overflow).

EOVERFLOW

The resulting file offset is a value that can't be represented correctly in an object of type `off_t`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_mknod()

Verify a client's ability to make a new filesystem entry point

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_mknod( resmgr_context_t *ctp,
                  io_mknod_t *msg,
                  iofunc_attr_t *attr,
                  iofunc_attr_t *dattr,
                  struct _client_info *info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_mknod_t](#) (p. 1292) structure that contains the message that the resource manager received; see below.

attr

NULL, or a pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the resource.

dattr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that you must set. The [iofunc_attr_t](#) structure describes the attributes of the parent directory.

info

NULL, or a pointer to a [_client_info](#) structure that contains information about the client. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_mknod()* helper function supports [mknod\(\)](#) (p. 1642) requests by verifying that the client can make a new filesystem entry point. It's similar to [iofunc_open\(\)](#) (p. 1321).

The *iofunc_mknod()* function checks to see if the client (described by the optional *info* structure) has access to open the resource (name passed in the *msg* structure). The *attr* structure describes the resource's attributes, and the optional *dattr* structure defines the attributes of the parent directory (i.e. if *dattr* isn't NULL, it implies that the resource identified by *attr* is being created within the directory specified by *dattr*).

The *info* argument can be passed as NULL, in which case *iofunc_mknod()* obtains the client information itself via a call to [iofunc_client_info_ext\(\)](#) (p. 1246). It is, of course, more efficient to get the client info once, rather than calling this function with NULL every time.

If an error occurs, the function returns information about a client's connection in *info* and a constant.

io_mknod_t structure

The *io_mknod_t* structure holds the `_IO_CONNECT` message received by the resource manager:

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_mknod_t;
```

This message structure is a union of an input message (coming to the resource manager), [_io_connect](#) (p. 1189), and two possible output or reply messages (going back to the client):

- [_io_connect_link_reply](#) (p. 1196) if the reply is redirecting the client to another resource

Or:

- [_io_connect_ftype_reply](#) (p. 1195) if the reply consists of a status and a file type.

Returns:

EOK

Success.

EBADFSYS

NULL was passed in *dattr*.

EFAULT

A fault occurred when the kernel tried to access the *info* buffer.

EINVAL

The client process is no longer valid.

ENOSYS

NULL was passed in *info*.

EPERM

The group ID or owner ID didn't match.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_mmap()

Handle an `_IO_MMAP` message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_mmap ( resmgr_context_t * hdr,
                  io_mmap_t * msg,
                  iofunc_ocb_t * ocb,
                  iofunc_attr_t * attr );
```

Arguments:

hdr

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_mmap_t](#) (p. 1295) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_mmap()* helper function provides functionality for the `_IO_MMAP` message. The `_IO_MMAP` message is an outcall from the Memory Manager (a part of the microkernel's `procnto`).

Note that if the Process Manager is to be able to execute from this resource, then you must use the *iofunc_mmap()* function.

io_mmap_t structure

The *io_mmap_t* structure holds the *_IO_MMAP* message received by the resource manager:

```
struct _io_mmap {
    uint16_t      type;
    uint16_t      combine_len;
    uint32_t      prot;
    uint64_t      offset;
    struct _msg_info info;
    uint32_t      required_prot;
    uint32_t      zero[5];
};

struct _io_mmap_reply {
    uint32_t      zero;
    uint32_t      allowed_prot;
    uint64_t      offset;
    int32_t      coid;
    int32_t      fd;
};

typedef union {
    struct _io_mmap      i;
    struct _io_mmap_reply o;
} io_mmap_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type *_io_mmap* that contains the following members:

type

_IO_MMAP.

combine_len

If the message is a combine message, *_IO_COMBINE_FLAG* is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

prot

The set of protection bits that *procnto* would *like* to have for the memory-mapped file. This can be a combination of at least the following protection bits, as defined in *<sys/mman.h>*:

- *PROT_EXEC* — the region can be executed.
- *PROT_NOCACHE* — disable caching of the region (e.g., so it can be used to access dual-ported memory).
- *PROT_NONE* — the region can't be accessed.
- *PROT_READ* — the region can be read.
- *PROT_WRITE* — the region can be written.

See the *required_prot* field, below.

offset

The offset into shared memory of the location that the client wants to start mapping.

info

A pointer to a [_msg_info](#) (p. 1731), structure that contains information about the message received by the resource manager.

required_prot

The set of permission bits that *procnto* *must* have for the memory-mapped file. Typically the difference between *prot* and *required_prot* is that *prot* may specify `PROT_WRITE` while *required_prot* omits it. This means that *procnto* would like to have write permissions on the file, but it's OK if it isn't present.

The resource manager can tell whether a new or old *procnto* is talking to it by looking at the *required_prot* field. If it's nonzero, it's a new *procnto*, and the code should pay attention to the field. If *required_prot* is zero, it's an old *procnto*, and the code should use *prot* as the required permissions.

The *o* member of the `io_mmap_t` structure is a structure of type `_io_mmap_reply` that contains the following members:

allowed_prot

The allowed protections for the file: `PROT_READ` is always present, `PROT_WRITE` is present if the file is opened for writing, and `PROT_EXEC` is present if the client process that opened the file has execute permissions specified for the file.

offset

Reserved for future use.

coid

A file descriptor that the process manager can use to access the mapped file.

fd

Reserved for future use.

Returns:

A nonpositive value (i.e., 0)

Successful completion.

EROFS

An attempt was made to memory map (mmap) a read-only file, using the PROT_WRITE page protection mode.

EACCES

The client doesn't have the appropriate permissions.

ENOMEM

Insufficient memory exists to allocate internal resources required to effect the mapping.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_mmap_default()

Default handler for _IO_MMAP messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_mmap_default ( resmgr_context_t * hdr,
                          io_mmap_t * msg,
                          iofunc_ocb_t * ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_mmap_t](#) (p. 1295) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_mmap()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_mmap_default()* function provides functionality for the `_IO_MMAP` message. This message is private to the Memory Manager (a part of the microkernel's `procnto`).

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *mmap* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all of the functions to their default values.

Note that if the Process Manager is to be able to execute from this resource, then you must use the [iofunc_mmap\(\)](#) (p. 1294) function.

The [iofunc_mmap_default\(\)](#) function calls [iofunc_mmap\(\)](#) (p. 1294) to do the actual work.

Returns:

A nonpositive value (i.e. 0)

Successful completion.

EROFS

An attempt to memory map (mmap) a read-only file, using the PROT_WRITE page protection mode.

EACCES

The client doesn't have the appropriate permissions.

ENOMEM

Insufficient memory exists to allocate internal resources required to effect the mapping.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_notify()

Install, poll, or remove a notification handler

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_notify( resmgr_context_t *ctp,
                  io_notify_t *msg,
                  iofunc_notify_t *nop,
                  int trig,
                  const int *notifycounts,
                  int *armed );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_notify_t](#) (p. 1302) structure that contains the message that the resource manager received; see below.

nop

An array of [iofunc_notify_t](#) (p. 1303) structures that represent the events supported by the calling resource manager. Traditionally this array contained three members which represent, in order, the input, output, and out-of-band notification lists. Since the addition of extended events (see below), three is now the minimum size of this array. The actual size must support indexing by the conditions being triggered up to `_NOTIFY_MAXCOND`.

Generally, this structure is maintained by the resource manager within an extended attributes structure.

trig

A bitmask indicating which sources are currently satisfied, and could cause a trigger to occur. This bitmask may be indicated via two sets of flags. Traditionally, the value was any combination of `_NOTIFY_COND_INPUT`, `_NOTIFY_COND_OUTPUT` and `_NOTIFY_COND_OBAND`, but you can set `_NOTIFY_COND_EXTEN` so that you can specify the following extended events:

- `_NOTIFY_CONDE_RDNORM` — normal data is available.
- `_NOTIFY_CONDE_WRNORM` — room for normal data.
- `_NOTIFY_CONDE_RDBAND` — out-of-band data is available.
- `_NOTIFY_CONDE_PRI` — priority data is available.
- `_NOTIFY_CONDE_WRBAND` — room for OOB data.
- `_NOTIFY_CONDE_ERR` — an error occurred on the device or stream.
- `_NOTIFY_CONDE_HUP` — the device has been disconnected.
- `_NOTIFY_CONDE_NVAL` — the file descriptor is invalid.

Note that the following flags are considered equivalent:

```
_NOTIFY_COND_INPUT  == _NOTIFY_CONDE_RDNORM
_NOTIFY_COND_OUTPUT == _NOTIFY_CONDE_WRNORM
_NOTIFY_COND_OBAND  == _NOTIFY_CONDE_RDBAND
```

Setting the `_NOTIFY_COND_EXTEN` flag affects the *armed* parameter, as described below.

You typically set this value, based on the conditions in effect at the time of the call.

notifycounts

NULL, or an array of integers representing the number of elements that must be present in the queue of each event represented by the *nop* array in order for the event to be triggered. Both this array and the *nop* array should contain the same number of elements. Note that if any condition is met, nothing is armed. Only if none of the conditions are met, does the event get armed in accordance with the *notifycounts* parameter. If this parameter is NULL, a value of 1 is assumed for all counts.

armed

NULL, or a pointer to a location where the function can store a 1 to indicate that a notification entry is armed, or a 0 otherwise.

If you set the `_NOTIFY_COND_EXTEN` bit in the *trig* argument, and *armed* isn't NULL, then when you call *iofunc_notify()*, *armed* must contain the number of elements in the *nop* and *notifycounts* arrays (provided *notifycounts* isn't NULL). Otherwise, the function assumes there are three elements in the *nop* and *notifycounts* arrays.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The POSIX layer helper function *iofunc_notify()* is used by a resource manager to implement notification.

This routine examines the message that the resource manager received (passed in the *msg* argument), and determines what action the client code is attempting to perform:

`_NOTIFY_ACTION_POLL`

Return a one-part IOV with the *flags* field set to indicate which conditions (input, output, or out-of-band) are available. The caller should return ([_RESMGR_NPARTS](#) (p. 2632)(1)) to the resource manager library, which returns a one-part message to the client.

`_NOTIFY_ACTION_POLLARM`

Similar to `_NOTIFY_ACTION_POLL`, with the additional characteristic of arming the event if *none* of the conditions is met.

`_NOTIFY_ACTION_TRANARM`

For each of the sources specified, create a notification entry and store the client's `struct sigevent` event structure in it. Note that only one transition arm is allowed at a time *per device*. If the client specifies an event of `SIGEV_NONE`, the action is to disarm. When the event is triggered, the notification is automatically disarmed.

`io_notify_t` structure

The `io_notify_t` structure holds the `_IO_NOTIFY` message received by the resource manager:

```
struct _io_notify {
    uint16_t          type;
    uint16_t          combine_len;
    int32_t           action;
    int32_t           flags;
    struct sigevent    event;
};

struct _io_notify_reply {
    uint32_t          zero;
    uint32_t          flags;
};

typedef union {
    struct _io_notify    i;
    struct _io_notify_reply o;
} io_notify_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type `_io_notify` that contains the following members:

type

`_IO_NOTIFY.`

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

action

`_NOTIFY_ACTION_POLL`, `_NOTIFY_ACTION_POLLARM`, or `_NOTIFY_ACTION_TRANARM`, as described above.

flags

One of the following:

- `_NOTIFY_COND_INPUT` — this condition is met when there are one or more units of input data available (i.e. clients can now issue reads).
- `_NOTIFY_COND_OUTPUT` — this condition is met when there's room in the output buffer for one or more units of data (i.e. clients can now issue writes).
- `_NOTIFY_COND_OBAND` — the condition is met when one or more units of out-of-band data are available.

event

A pointer to a [sigevent](#) (p. 2961) structure that defines the event that the resource manager is to deliver once a condition is met.

The *o* member is a structure of type `_io_notify_reply` that contains the following members:

flags

Which of the conditions were triggered; see the *flags* for `_io_notify`, above.

iofunc_notify_t structure

The `iofunc_notify_t` structure is defined in `<sys/iofunc.h>` as follows:

```
typedef struct _iofunc_notify {
    int cnt;
    struct _iofunc_notify_event *list;
} iofunc_notify_t;
```

The members of the `iofunc_notify_t` structure include:

cnt

The smallest *cnt* member in the list; see below.

list

A pointer to a linked list of `iofunc_notify_event_t` structures that represent (in order), the input, output, and out-of-band notification lists.

The `iofunc_notify_event_t` structure is defined as:

```
typedef struct _iofunc_notify_event {
    struct _iofunc_notify_event *next;
    int rcvid;
    int scoid;
    int cnt;
    struct sigevent event;
} iofunc_notify_event_t;
```

The members of the `iofunc_notify_event_t` structure include:

next

A pointer to the next element in the list.

rcvid

The receive ID of the client to notify.

scoid

The server connection ID.

cnt

The number of bytes available. Some clients, such as `io-char`, may want a sufficiently large amount of data to be available before they access it.

event

A pointer to a [sigevent](#) (p. 2961) structure that defines the event that the resource manager is to deliver once a condition is met.

The `sys/iofunc.h` file also defines the following macros that work with the arrays of `iofunc_notify_t` structures:

```
#define IOFUNC_NOTIFY_DISARM(__nop, __index) ...
```

Disarm the list specified by `__index` in `__nop`.

```
#define IOFUNC_NOTIFY_INIT(__nop) ...
```

Initialize the three lists in `__nop`.

Returns:

-1

Success; the resource manager library should return a one-part IOV to the client.

EBUSY

A notification was already armed for this resource, and this library function enforces a restriction of one per resource.

Examples:

See *Writing a Resource Manager*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_notify_remove()

Remove notification entries from list

Synopsis:

```
#include <sys/iofunc.h>

void iofunc_notify_remove( resmgr_context_t * ctp,
                          iofunc_notify_t * nop );
```

Arguments:

ctp

NULL, or a pointer to a [resmgr_context_t](#) (p. 2598) structure for the client whose entries you want to remove.

nop

An array of three [iofunc_notify_t](#) (p. 1303) structures that represent (in order), the input, output, and out-of-band notification lists whose entries you want to remove; for information about this structure, see the documentation for *iofunc_notify()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_notify_remove()* function removes all of the entries associated with the current client from the notification list passed in *nop*. The client information is obtained from the *ctp*.

If the *ctp* pointer is NULL, then *all* of the notify entries will be removed. A resource manager generally calls this function, with NULL as the *ctp* in the *close_ocb* callout, to clean up all handles associated with this connection. If the handles are shared between several connections, then the *ctp* should be provided to clean up after each client.

Examples:

See *Writing a Resource Manager*.

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_notify_trigger()

Send notifications to queued clients

Synopsis:

```
#include <sys/iofunc.h>

void iofunc_notify_trigger( iofunc_notify_t *nop,
                           int count,
                           int index );
```

Arguments:

nop

An array of three [iofunc_notify_t](#) (p. 1303) structures that represent (in order), the input, output, and out-of-band notification lists whose entries you want to examine; for information about this structure, see the documentation for [iofunc_notify\(\)](#).

count

The count that you want to compare to the trigger value for the event.

index

The index into the *nop* array that you want to check; one of the following:

- IOFUNC_NOTIFY_INPUT
- IOFUNC_NOTIFY_OUTPUT
- IOFUNC_NOTIFY_OBAND

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_notify_trigger()* function examines all entries given in the list maintained at *nop [index]* to see if the event should be delivered to the client. If the specified *count* is greater than or equal to the trigger count for the particular notification list element, this function calls [MsgDeliverEvent\(\)](#) (p. 1736) to deliver the event to the client whose *rcvid* is stored in the notification list element, and the list element is disarmed.

Note that if the client has specified a code of `SI_NOTIFY`, then the value that the client specified (e.g. the *value* member of the struct `sigevent`) has the top three bits ORed with the reason for the trigger (this is the expression `_NOTIFY_COND_INPUT << index`), as in the following table:

index = IOFUNC_NOTIFY_INPUT

0x10000000, or `_NOTIFY_COND_INPUT`

index = IOFUNC_NOTIFY_OUTPUT

0x20000000, or `_NOTIFY_COND_OUTPUT`

index = IOFUNC_NOTIFY_OBAND

0x40000000, or `_NOTIFY_COND_OBAND`

If the client has specified a code of something other than `SI_NOTIFY` then this routine doesn't modify the value member in any way.

The `sys/iofunc.h` file also defines the `IOFUNC_NOTIFY_INPUT_CHECK()`, `IOFUNC_NOTIFY_OUTPUT_CHECK()`, and `IOFUNC_NOTIFY_OBAND_CHECK()` macros, which you can use to see if there are enough bytes in the lists to make it worthwhile to call `iofunc_notify_trigger()` to notify clients about:

```
#define IOFUNC_NOTIFY_INPUT_CHECK(__nop, __cnt, __tran) ...
#define IOFUNC_NOTIFY_OUTPUT_CHECK(__nop, __cnt) ...
#define IOFUNC_NOTIFY_OBAND_CHECK(__nop, __cnt, __tran) ...
```

The arguments are:

__nop

An array of three `iofunc_notify_t` (p. 1303) structures that represent (in order), the input, output, and out-of-band notification lists whose entries you want to examine.

__cnt

The count that you want to compare to the trigger value for the event.

__tran

(`IOFUNC_NOTIFY_INPUT_CHECK()` and `IOFUNC_NOTIFY_OBAND_CHECK()` only) Nonzero if you want to arm the trigger when the list makes the transition from empty to nonempty. (For example, `mq_notify()` (p. 1707) does this.)

Examples:

See *Writing a Resource Manager*

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_ocb_attach()

Initialize an Open Control Block

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_ocb_attach(
    resmgr_context_t * ctp,
    io_open_t * msg,
    iofunc_ocb_t * ocb,
    iofunc_attr_t * attr,
    const resmgr_io_funcs_t * io_funcs );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_open_t](#) (p. 1323) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_open()*.

ocb

NULL, or a pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to a [iofunc_attr_t](#) (p. 1221) structure that defines the characteristics of the device that the resource manager handles.

io_funcs

A pointer to a [resmgr_io_funcs_t](#) (p. 2610) that specifies the I/O functions for the resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_ocb_attach()* function examines the mode specified by the *io_open msg*, and increments the read and write count flags (*ocb->attr->rcount* and *ocb->attr->wcount*), and the locking flags (*ocb->attr->rlocks* and *ocb->attr->wlocks*), as specified by the open mode. This function then calls [resmgr_open_bind\(\)](#) (p. 2636) to bind the newly created OCB to the request.

The *iofunc_ocb_attach()* function is called by [iofunc_open_default\(\)](#) (p. 1325) as part of its initialization.

This function allocates the memory for the OCB if you pass NULL as the *ocb*.

Returns:

EOK

Successful completion.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_ocb_alloc()

Allocate an iofunc Open Control Block

Synopsis:

```
#include <sys/iofunc.h>

iofunc_ocb_t * iofunc_ocb_alloc(
    resmgr_context_t * ctp,
    iofunc_attr_t * attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

attr

A pointer to a [iofunc_attr_t](#) (p. 1221) structure that defines the characteristics of the device that the resource manager handles.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_ocb_alloc()* function allocates an iofunc OCB. It has a number of uses:

- It can be used as a helper function to encapsulate the allocation of the iofunc OCB, so that your routines don't have to know the details of the iofunc OCB structure.
- Because it's in the resource manager shared library, you can override this function with your own, allowing you to manage an OCB that has additional members, perhaps specific to your particular resource manager. If you do this, be sure to place the iofunc OCB structure as the first element of your extended OCB, and also override the [iofunc_ocb_free\(\)](#) (p. 1317) function to release memory.
- Another reason to override *iofunc_ocb_alloc()* might be to place limits on the number of OCBs that are in existence at any one time; the current function simply allocates OCBs until the free store is exhausted.

You should fill in the attribute's mount structure (i.e. the *attr->mount* pointer) instead of replacing this function.



If you specify *ocb_alloc* and *ocb_free* callouts in the attribute's mount structure, then you should use the callouts instead of calling the standard *iofunc_ocb_alloc()* and *iofunc_ocb_free()* functions.

Returns:

A pointer to an [iofunc_ocb_t](#) (p. 1319) OCB structure, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

ENFILE

There are too many open files in the system. For *iofunc_ocb_alloc()*, this happens if there isn't enough memory to allocate the OCB; if you override this function, your replacement might place further restrictions on the number of OCBs.

Examples:

Override *iofunc_ocb_alloc()* and *iofunc_ocb_free()* to manage an extended OCB:

```
typedef struct
{
    iofunc_ocb_t    iofuncOCB; /* the OCB used by iofunc_* */
    int             myFlags;
    char            moreOfMyStuff;
} MyOCBT;

MyOCBT *iofunc_ocb_alloc (resmgr_context_t *ctp,
                        iofunc_attr_t *attr)
{
    return ((MyOCBT *) calloc (1, sizeof (MyOCBT)));
}

void iofunc_ocb_free (MyOCBT *ocb)
{
    free (ocb);
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_ocr_detach()

Release Open Control Block resources

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_ocr_detach( resmgr_context_t * ctp,
                      iofunc_ocr_t * ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ocr

A pointer to the [iofunc_ocr_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *iofunc_ocr_detach()* function releases any resources allocated to the passed *ocr*, such as any memory map (mmap) entries.



This function doesn't free the memory associated with the OCB itself.

The *iofunc_ocr_detach()* function also updates the time structure, by calling [iofunc_time_update\(\)](#) (p. 1364), and decrements the read, write, lock, and use counters, according to the mode that was used to open the resource (from *ocr->ioflag*).

The counters are incremented in [iofunc_ocr_attach\(\)](#) (p. 1311), and represent the number of OCBs that are using the managed resource in the respective manners (e.g.: *ocr->attr->rcount* keeps count of how many OCBs are using the resource specified by *attr* for read access).

If you're are using [iofunc_mmap\(\)](#) (p. 1294) or [iofunc_mmap_default\(\)](#) (p. 1298), you must call [iofunc_ocb_detach\(\)](#) to clean up. This function is called by [iofunc_close_ocb\(\)](#) (p. 1253).

Returns:

A bitwise OR of flags describing the state of the managed resource:

IOFUNC_OCB_LAST_READER

This OCB was the last one performing read operations on the resource. This flag is set when the `ocb->attr->rcount` flag is decremented to zero.

IOFUNC_OCB_LAST_WRITER

This OCB was the last one performing write operations on the resource. This flag is set when the `ocb->attr->>wcount` flag is decremented to zero.

IOFUNC_OCB_LAST_RDLOCK

This OCB was the last one holding a read lock on the resource. This flag is set when the `ocb->attr->rlocks` flag is decremented to zero.

IOFUNC_OCB_LAST_WRLOCK

This OCB was the last one holding a write lock on the resource. This flag is set when the `ocb->attr->>wlocks` flag is decremented to zero.

IOFUNC_OCB_LAST_INUSE

This OCB was the last one using the resource. This flag is set when the `ocb->attr->count` flag is decremented to zero.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_ocb_free()

Deallocate an iofunc Open Control Block's memory

Synopsis:

```
#include <sys/iofunc.h>

void iofunc_ocb_free( iofunc_ocb_t * ocb );
```

Arguments:

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_ocb_free()* function returns the memory allocated to an iofunc OCB to the free store pool. This function is the complement of [iofunc_ocb_alloc\(\)](#) (p. 1313).

If you've overridden the definition of *iofunc_ocb_alloc()*, you should also override the definition of *iofunc_ocb_free()* to correctly handle the release of the memory. This is because the *iofunc_ocb_alloc()* function uses an internal memory management function to allocate the memory, and the default *iofunc_ocb_free()* function also uses this internal function to deallocate memory. Therefore, you can't mix internal memory management functions (*_salloc()* and *_sfree()*) with user-level memory management functions (*calloc()* and *free()*).

You should fill in the attribute's mount structure (i.e. the *attr->mount* pointer) instead of replacing this function.



If you specify *iofunc_ocb_free()* and *iofunc_ocb_alloc()* callouts in the attribute's mount structure, then you should use the callouts instead of calling the standard *iofunc_ocb_free()* and *iofunc_ocb_alloc()* functions.

Examples:

See [iofunc_ocb_alloc\(\)](#) (p. 1313).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_ocb_t

Open Control Block structure

Synopsis:

```
#include <sys/iofunc.h>

typedef struct _iofunc_ocb {
    IOFUNC_ATTR_T            *attr;
    int32_t                  ioflag;
#if !defined(_IOFUNC_OFFSET_BITS) || _IOFUNC_OFFSET_BITS == 64

    #if _FILE_OFFSET_BITS - 0 == 64
        off_t                offset;
    #else
        off64_t              offset;
    #endif
#elif _IOFUNC_OFFSET_BITS - 0 == 32
    #if !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
        #if defined(__LITTLEENDIAN__)
            off_t              offset;
            off_t              offset_hi;
        #elif defined(__BIGENDIAN__)
            off_t              offset_hi;
            off_t              offset;
        #else
            #error endian not configured for system
        #endif
    #else
        #if defined(__LITTLEENDIAN__)
            int32_t            offset;
            int32_t            offset_hi;
        #elif defined(__BIGENDIAN__)
            int32_t            offset_hi;
            int32_t            offset;
        #else
            #error endian not configured for system
        #endif
    #endif
#else
    #error _IOFUNC_OFFSET_BITS value is unsupported
#endif
    uint16_t                 sflag;
    uint16_t                 flags;
    void                     *reserved;
} iofunc_ocb_t;
```

Description:

The `iofunc_ocb_t` structure is an *Open Control Block*, a block of data that's established by a resource manager during its handling of the client's `open()` function.

A resource manager creates an instance of this structure whenever a client opens a resource. For example, [iofunc_open_default\(\)](#) (p. 1325) calls [iofunc_ocb_alloc\(\)](#) (p. 1313) to allocate an OCB. The OCB exists until the client closes the file descriptor associated

with the open operation. The resource manager passes this structure to all of the functions that implement the I/O operations for the file descriptor.

The `iofunc_ocb_t` structure includes the following members:

attr

A pointer to the OCB's attributes. By default, this structure is of type `iofunc_attr_t` (p. 1221), but you can redefine the `IOFUNC_ATTR_T` manifest if you want to use a different structure in your resource manager.

ioflag

The mode (e.g. reading, writing, blocking) that the resource was opened with. The open modes (as passed to `open()` on the client side) correspond to the `ioflag` values as follows:

Open mode	<i>ioflag</i> value
O_RDONLY	_IO_FLAG_RD
O_RDWR	_IO_FLAG_RD _IO_FLAG_WR
O_WRONLY	_IO_FLAG_WR

This information is inherited from the `io_connect_t` structure that's available in the message passed to the open handler.

offset, offset_hi

The read/write offset into the resource (e.g. our current `lseek()` (p. 1555) position within a file), defined in a variety of ways to suit 32- and 64-bit offsets. Your resource manager can modify this offset.

sflag

The sharing mode; see `sopen()` (p. 3083). This information is inherited from the `io_connect_t` structure that's available in the message passed to the open handler.

flags

When the `IOFUNC_OCB_PRIVILEGED` bit is set, a privileged process (i.e. root) performed the `open()`. Additionally, you can use flags in the range defined by `IOFUNC_OCB_FLAGS_PRIVATE` (see `<sys/iofunc.h>`) for your own purposes. Your resource manager can modify these flags.

Classification:

[QNX Neutrino](#) (p. 46)

iofunc_open()

Verify a client's ability to open a resource

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_open( resmgr_context_t *ctp,
                 io_open_t *msg,
                 iofunc_attr_t *attr,
                 iofunc_attr_t *dattr,
                 struct _client_info *info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_open_t](#) (p. 1323) structure that contains the message that the resource manager received; see below.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the resource.

dattr

NULL, or a pointer to the [iofunc_attr_t](#) structure that describes the characteristics of the parent directory.

info

NULL, or a pointer to a [_client_info](#) structure that contains the information about a client connection. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_open()* function checks to see if the client (described by the optional *info* structure) has access to open the resource whose name is passed in *msg->connect.path*.

The *attr* structure describes the resource's attributes. The optional *dattr* structure defines the attributes of the parent directory; if *dattr* isn't NULL, the resource identified by *attr* is being created within the directory specified by *dattr*.

The *info* argument can be passed as NULL, in which case *iofunc_open()* obtains the client information itself via a call to *iofunc_client_info_ext()* (p. 1246). It is, of course, more efficient to get the client info once, rather than calling this function with NULL every time.

Note that if you're handling a request to read directory entry, you must return data formatted to match the *struct dirent* (p. 422) type. A helper function, *iofunc_stat()* (p. 1353), can aid in this.

A resource manager's response to an *open()* request isn't always a yes-or-no answer. It's possible to return a connect message indicating that the server would like some other action taken. For example, if the open occurs on a path that represents a symbolic link to some other path, the server could respond using the *_IO_SET_CONNECT_RET()* macro and the *_IO_CONNECT_RET_LINK* value.

For example, an open handler that only redirects pathnames might look something like:

```
io_open(resmgr_context_t *ctp, io_open_t *msg,
        iofunc_attr_t *dattr, void *extra) {
    char *newpath;

    /* Do all the error/access checking ... */

    /* Lookup the redirected path and store
       the new path in 'newpath' */
    newpath = get_a_new_path(msg->connect.path);

    _IO_SET_CONNECT_RET(ctp, _IO_CONNECT_RET_LINK);
    len = strlen(newpath) + 1;

    msg->link_reply.eflag = msg->connect.eflag;
    msg->link_reply.nentries = 0;
    msg->link_reply.path_len = len;
    strcpy((char *) (msg->link_reply + 1), newpath);

    len += sizeof(msg->link_reply);

    return(_RESMGR_PTR(ctp, &msg->link_reply, len));
}
```

In this example, we use the macro *_IO_SET_CONNECT_RET()* (defined in *<sys/iomsg.h>*) to set the *ctp->status* field to *_IO_CONNECT_RET_LINK*. This value indicates to the resource-manager framework that the return value isn't actually a simple return code, but a new request to be processed.

The path for this new request follows directly after the *link_reply* structure and is *path_len* bytes long. The final few lines of the code just stuff an IOV with the reply

message (and the new path to be queried) and return to the resource-manager framework.

io_open_t structure

The `io_open_t` structure holds the `_IO_CONNECT` message received by the resource manager:

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_open_t;
```

This message structure is a union of an input message (coming to the resource manager), `_io_connect` (p. 1189), and two possible output or reply messages (going back to the client):

- `_io_connect_link_reply` (p. 1196) if the reply is redirecting the client to another resource

Or:

- `_io_connect_ftype_reply` (p. 1195) if the reply consists of a status and a file type.

Returns:

EOK

Successful completion.

Other

There was an error, as defined by the POSIX semantics for the open call. This error should be returned to the next higher level.

Examples:

This is a sample skeleton for a typical filesystem, in pseudo-code, to illustrate the steps that need to be taken to handle an open request for a file:

```
if the open request is for a path (i.e. multiple
    directory levels)
    call iofunc_client_info_ext to get information
    about client
    for each directory component
        call iofunc_check_access to check execute
        permission for access
        /*
        recall that execute permission on a
        directory is really the "search"
        permission for that directory
        */
    next
    /*
    at this point you have verified access
    to the target
    */
endif
```

```
if O_CREAT is set and the file doesn't exist
    call iofunc_open, passing the attribute of the
    parent as dattr
    if the iofunc_open succeeds,
        do the work to create the new inode,
        or whatever
    endif
else
    call iofunc_open, passing the attr of the file
    and NULL for dattr
endif

/*
    at this point, check for things like o_trunc,
    etc. -- things that you have to do for the attr
*/

call iofunc_ocb_attach
return EOK
```

For a device (i.e. *resmgr_attach()* didn't specify that the managed resource is a directory), the following steps apply:

```
/*
    at startup time (i.e.: in the main() of the
    resource manager)
*/
call iofunc_attr_init to initialize an attribute
structure

/* in the io_open message handler: */
call iofunc_open, passing in the attribute of the
device and NULL for dattr

call iofunc_ocb_attach
return EOK
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_open_default()

Default handler for `_IO_CONNECT` messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_open_default( resmgr_context_t *ctp,
                        io_open_t *msg,
                        iofunc_attr_t *attr,
                        void *extra );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_open_t](#) (p. 1323) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_open()*.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that defines the characteristics of the device that the resource manager is controlling.

extra

Extra information from the library. If you're calling *iofunc_open_default()* from a resource manager's *open()* function (see [resmgr_connect_funcs_t](#) (p. 2592)), simply pass the *extra* argument that's passed to *open()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_open_default()* function implements the default actions for the `_IO_CONNECT` message in a resource manager. This function calls:

- [iofunc_open\(\)](#) (p. 1321) to check the client's open mode against the resources attributes to see if the client can open the resource in that mode
- [iofunc_ocr_alloc\(\)](#) (p. 1313) to allocate an Open Control Block (OCB)
- [iofunc_ocr_attach\(\)](#) (p. 1311) to initialize the OCB. This function calls [resmgr_open_bind\(\)](#) (p. 2636) to bind the newly created OCB to the request.

You can place this function directly into the *connect_funcs* table passed to [resmgr_attach\(\)](#) (p. 2580), at the *open* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

See the “Examples” section in the description of [iofunc_open\(\)](#) (p. 1321) for the skeleton outline of the functionality (the second example, where [resmgr_attach\(\)](#) doesn't specify that the managed resource is a directory).

Returns:

EOK

Successful completion.

ENOSPC

There's insufficient memory to allocate the OCB.

ENOMEM

There's insufficient memory to allocate an internal data structure required by [resmgr_open_bind\(\)](#).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_openfd()

Increment count and locking flags

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_openfd( resmgr_context_t *ctp,
                  io_openfd_t *msg,
                  iofunc_ocb_t *ocb,
                  iofunc_attr_t *attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_openfd_t](#) (p. 1328) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_openfd()* helper function examines the mode specified by the `_IO_OPENFD` message, and increments the read and write count flags (*ocb->attr->rcount* and *ocb->attr->wcount*), and the locking flags (*ocb->attr->rlocks* and *ocb->attr->wlocks*), as specified by the open mode.

The function does what's needed to support the [openfd\(\)](#) (p. 1916) function.

io_openfd_t structure

The `io_openfd_t` structure holds the `_IO_OPENFD` message received by the resource manager:

```
struct _io_openfd {
    uint16_t          type;
    uint16_t          combine_len;
    uint32_t          ioflag;
    uint16_t          sflag;
    uint16_t          xtype;
    struct _msg_info  info;
    uint32_t          reserved2;
    uint32_t          key;
};

typedef union {
    struct _io_openfd  i;
} io_openfd_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_openfd` that contains the following members:

type

`_IO_OPENFD`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

ioflag

How the client wants to open the file; a combination of the following bits:

- `_IO_FLAG_RD` — permit the file to be read.
- `_IO_FLAG_WR` — permit the file to be written.
- `O_APPEND` — cause each record that's written to be written at the end of the file.
- `O_TRUNC` — if the file exists, truncate it to contain no data. This flag has no effect if the file doesn't exist.

sflag

How the client wants the file to be shared; a combination of the following bits:

- `SH_COMPAT` — set compatibility mode.

- SH_DENYRW — prevent read or write access to the file.
- SH_DENYWR — prevent write access to the file.
- SH_DENYRD — prevent read access to the file.
- SH_DENYNO — permit both read and write access to the file.

xtype

Extended type information that can change the behavior of an I/O function.
One of:

- _IO_OPENFD_NONE — no extended type information.
- _IO_OPENFD_PIPE — a pipe is being opened.
- _IO_OPENFD_RESERVED — reserved

info

A pointer to a [_msg_info](#) (p. 1731) structure that contains information about the message received by the resource manager.

key

Reserved for future use.

Returns:**EOK**

Success.

EACCES

You don't have permission to open the file.

EBUSY

The file has shared locks that are in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_openfd_default()

Default handler for _IO_OPENFD messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_openfd_default( resmgr_context_t *ctp,
                          io_openfd_t *msg,
                          iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_openfd_t](#) (p. 1328) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_openfd()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The function *iofunc_openfd_default()* function implements the default actions for `_IO_OPENFD` messages.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *openfd* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The `iofunc_openfd_default()` function calls [iofunc_openfd\(\)](#) (p. 1327) to do the actual work.

Returns:**EOK**

Success.

EACCES

You don't have permission to open the file.

EBUSY

The file has shared locks that are in use.

EINVAL

The message type is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_pathconf()

Support *pathconf()* requests

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_pathconf( resmgr_context_t *ctp,
                    io_pathconf_t *msg,
                    iofunc_ocb_t *ocb,
                    iofunc_attr_t *attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_pathconf_t](#) (p. 1333) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_pathconf()* helper function does what's needed to support [pathconf\(\)](#) (p. 1936) with the *mount* and *attr* passed to it. Other *fsys_pathconf()* requests need to be handled by the caller.

If you write your own *pathconf* callout for your resource manager, use the following macro to pass the requested value back to the caller:

```
_IO_SET_PATHCONF_VALUE( resmgr_context_t *ctp,
                        int value )
```

io_pathconf_t structure

The `io_pathconf_t` structure holds the `_IO_PATHCONF` message received by the resource manager:

```
struct _io_pathconf {
    uint16_t          type;
    uint16x_t         combine_len;
    short             name;
    uint16_t          zero;
};

typedef union {
    struct _io_pathconf i;
    /* value is returned with MsgReply */
} io_pathconf_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_pathconf` that contains the following members:

type

`_IO_PATHCONF.`

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

name

The name of the configurable limit; see [pathconf\(\)](#) (p. 1936).

Returns:

EOK, or `_RESMGR_DEFAULT` if the function didn't handle the *pathconf()* request.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

iofunc_pathconf_default()

Default handler for `_IO_PATHCONF` messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_pathconf_default( resmgr_context_t *ctp,
                           io_pathconf_t *msg,
                           iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_pathconf_t](#) (p. 1333) structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_pathconf()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_pathconf_default()* function implements POSIX semantics for the client's *pathconf()* call, which is received as an `_IO_PATHCONF` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *pathconf* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all of the functions to their default values.

The *iofunc_pathconf_default()* function returns information about the resource, as per the POSIX specifications for *pathconf()*. The *iofunc_pathconf_default()* function simply calls *iofunc_pathconf()* (p. 1332) to do the actual work.

Returns:**EOK**

Successful completion.

EINVAL

The pathconf parameter being ascertained wasn't one of
_PC_CHOWN_RESTRICTED, _PC_NO_TRUNC, or _PC_SYNC_IO.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_read_default()

Default handler for `_IO_READ` messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_read_default( resmgr_context_t *ctp,
                        io_read_t *msg,
                        iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_read_t](#) (p. 1340) structure that contains the message that the resource manager received. For more information, see the documentation for [iofunc_read_verify\(\)](#).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [iofunc_read_default\(\)](#) function implements POSIX semantics for the client's `read()` call, which is received as an `_IO_READ` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the `io_funcs` argument to [resmgr_attach\(\)](#) (p. 2580), at the `read` position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The *iofunc_read_default()* function calls *iofunc_read_verify()* (p. 1339) to check the permissions on the file. If the permissions are appropriate, *iofunc_read_default()* indicates that it read zero bytes.

Returns:**EBADF**

The client doesn't have read access to this resource.

EINVAL

The extended type information is invalid.

EOK

The client has read access to this resource.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_read_verify()

Verify a client's read access to a resource

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_read_verify( resmgr_context_t* ctp,
                       io_read_t* msg,
                       iofunc_ocb_t* ocb,
                       int* nonblock );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_read_t](#) (p. 1340) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

nonblock

NULL, or a pointer to a location where the function can store a value that indicates whether or not the device is nonblocking:

- Nonzero — the client doesn't want to be blocked (i.e. O_NONBLOCK was set).
- Zero — the client wants to be blocked.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_read_verify()* helper function checks that the client that sent the `_IO_READ` message actually has read access to the resource, and, if *nonblock* isn't NULL, sets *nonblock* to `O_NONBLOCK` or 0).

The read permission check is done against *ocb->ioflag*.

Note that the `io_read_t` message has an override flag called *msg->i.xtype*. This flag allows the client to override the default blocking behavior for the resource on a per-request basis. This override flag is checked, and returned in the optional *nonblock*.

Note that if you're reading from a directory entry, you must return `struct dirent` (p. 422) structures in the *read* callout for your resource manager.

You'll also need to indicate how many bytes were read. You can do this with the macro:

```
_IO_SET_READ_NBYTES( resmgr_context_t *ctp,  
                     int nbytes )
```

`io_read_t` structure

The `io_read_t` structure holds the `_IO_READ` message received by the resource manager:

```
struct _io_read {  
    uint16_t          type;  
    uint16_t          combine_len;  
    int32_t           nbytes;  
    uint32_t          xtype;  
    uint32_t          zero;  
};  
  
typedef union {  
    struct _io_read      i;  
    /* unsigned char      data[nbytes]; */  
    /* nbytes is returned with MsgReply */  
} io_read_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_read` that contains the following members:

type

`_IO_READ`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

nbytes

The number of bytes that the client wants to read.

xtype

Extended type information; one of:

- `_IO_XTYPE_NONE`
- `_IO_XTYPE_READCOND`
- `_IO_XTYPE_QUEUE`
- `_IO_XTYPE_TCPIP`
- `_IO_XTYPE_TCPIP_MSG`
- `_IO_XTYPE_OFFSET`
- `_IO_XTYPE_REGISTRY`
- `_IO_XFLAG_DIR_EXTRA_HINT` — this flag is valid only when reading from a directory. The filesystem should normally return extra directory information when it's easy to get. If this flag is set, it is a hint to the filesystem to try harder (possibly causing media lookups) to return the extra information. The most common use would be to return `_DTYPE_LSTAT` information.
- `_IO_XFLAG_NONBLOCK`
- `_IO_XFLAG_BLOCK`

For more information, see “Handling other read/write details” in the Handling Read and Write Messages chapter of *Writing a Resource Manager*.

The commented-out declaration for *data* indicates that *nbytes* bytes of data immediately follow the `io_read_t` structure.

Returns:**EOK**

The client has read access to this resource.

EBADF

The client doesn't have read access to this resource.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

iofunc_readlink()

Verify a client's ability to read a symbolic link

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_readlink( resmgr_context_t *ctp,
                    io_readlink_t *msg,
                    iofunc_attr_t *attr,
                    struct _client_info *info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_readlink_t](#) (p. 1344) structure that contains the message that the resource manager received; see below.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

info

A pointer to a [_client_info](#) structure that contains the information about a client connection. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_readlink()* helper function supports [readlink\(\)](#) (p. 2520) requests by verifying that the client can read a symbolic link. It's similar to [iofunc_open\(\)](#) (p. 1321).

The *iofunc_readlink()* function checks to see if the client (described by the optional *info* structure) has access to open the resource (name passed in the *msg* structure). The *attr* structure describes the resource's attributes.

The *info* argument can be passed as *NULL*, in which case *iofunc_readlink()* obtains the client information itself via a call to *iofunc_client_info_ext()* (p. 1246). It is, of course, more efficient to get the client info once, rather than calling this function with *NULL* every time.

The *iofunc_readlink()* function handles the readlink verification for the POSIX layer.

io_readlink_t structure

The *io_readlink_t* structure holds the *_IO_CONNECT* message received by the resource manager:

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_readlink_t;
```

This message structure is a union of an input message (coming to the resource manager), *_io_connect* (p. 1189), and two possible output or reply messages (going back to the client):

- *_io_connect_link_reply* (p. 1196) if the reply is redirecting the client to another resource
- Or:
- *_io_connect_ftype_reply* (p. 1195) if the reply consists of a status and a file type.

Returns:

EBADFSYS

NULL was passed in *attr*.

EOK

Successful completion.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

iofunc_rename()

Do permission checks for a _IO_CONNECT_RENAME message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_rename( resmgr_context_t* ctp,
                  io_rename_t* msg,
                  iofunc_attr_t* oldattr,
                  iofunc_attr_t* olddattr,
                  iofunc_attr_t* newattr,
                  iofunc_attr_t* newdattr,
                  struct _client_info* info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_rename_t](#) (p. 1347) structure that contains the message that the resource manager received; see below.

oldattr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the resource.

olddattr

NULL, or a pointer to the [iofunc_attr_t](#) structure that describes the characteristics of the parent directory.

newattr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the target, if it exists.

newdattr

NULL, or a pointer to the [iofunc_attr_t](#) structure that describes the characteristics of the parent directory of the target.

info

NULL, or a pointer to a `_client_info` structure that contains the information about a client connection. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `iofunc_rename()` does permission checks for the `_IO_CONNECT` message (subtype `_IO_CONNECT_RENAME`) for context `ctp`. The `newattr` argument is the attribute of the target if it already exists.

This function is similar to [iofunc_open\(\)](#) (p. 1321). The `iofunc_rename()` function checks to see if the client (described by the optional `info` structure) has access to open the resource (name passed in the `msg` structure). The `attr` structure describes the resource's attributes.

The `info` argument can be passed as `NULL`, in which case `iofunc_rename()` obtains the client information itself via a call to [iofunc_client_info_ext\(\)](#) (p. 1246). It is, of course, more efficient to get the client information once, rather than call this function with `NULL` every time.

io_rename_t structure

The `io_rename_t` structure holds the `_IO_CONNECT` message received by the resource manager:

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_rename_t;
```

This message structure is a union of an input message (coming to the resource manager), [_io_connect](#) (p. 1189), and two possible output or reply messages (going back to the client):

- [_io_connect_link_reply](#) (p. 1196) if the reply is redirecting the client to another resource

Or:

- [_io_connect_ftype_reply](#) (p. 1195) if the reply consists of a status and a file type.

The reply includes the following extra information:

```
typedef union _io_rename_extra {
```

```
char path[1];  
} io_rename_extra_t;
```

Returns:**EACCES**

The client doesn't have permissions to do the operation.

EBADFSYS

NULL was passed in *oldattr*, *olddattr*, or *newdattr*.

EFAULT

A fault occurred when the kernel tried to access the *info* buffer.

EINVAL

The *oldattr* and *newdattr* have identical values, the client process is no longer valid, or attempt to remove the parent (".") directory.

EISDIR

The old link is a directory but the new link isn't a directory.

ENOTDIR

Attempt to unlink a nondirectory entry using directory semantics (e.g. `rmdir file`).

ENOTEMPTY

Attempt to remove a directory that isn't empty.

EOK

Successful completion or there was already a *newattr* entry.

EPERM

The group ID or owner ID didn't match.

EROFS

Attempt to remove an entry on a read-only filesystem.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

iofunc_space_verify()

Do permission checks for the _IO_SPACE message

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_space_verify( resmgr_context_t *ctp,
                        io_space_t *msg,
                        iofunc_ocb_t *ocb,
                        int *nonblock );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_space_t](#) (p. 1351) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

nonblock

NULL, or a pointer to a location where the function can store a value that indicates whether or not the device is nonblocking:

- Zero — the client doesn't want to be blocked (i.e. `O_NONBLOCK` was set).
- Nonzero — the client wants to be blocked.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_space_verify()* helper function checks the client's permission for an `_IO_SPACE` message.

io_space_t structure

The `io_space_t` structure holds the `_IO_SPACE` message received by the resource manager:

```
struct _io_space {
    uint16_t      type;
    uint16_t      combine_len;
    uint16_t      subtype;
    short         whence;
    uint64_t      start;
    uint64_t      len;
};

typedef union {
    struct _io_space    i;
    uint64_t            o;
} io_space_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type `_io_space` that contains the following members:

type

`_IO_SPACE`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

subtype

`F_ALLOCS` or `F_FREES`.

whence

The position in the file. The possible values (defined in `<unistd.h>`) are:

SEEK_CUR

The new file position is computed relative to the current file position. The value of *start* may be positive, negative or zero.

SEEK_END

The new file position is computed relative to the end of the file.

SEEK_SET

The new file position is computed relative to the start of the file.
The value of *start* must not be negative.

start

The relative offset from the file position determined by the *whence* member.

len

The relative size by which to increase the file.

A value of zero means to end of file.

The *o* member is the file size.

Returns:**EBADF**

The client doesn't have read access to this resource.

EISDIR

The resource is a directory.

EOK

The client has read access to this resource.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_stat()

Populate a stat structure

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_stat( resmgr_context_t* ctp,
                iofunc_attr_t* attr,
                struct stat* stat );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

stat

A pointer to the `stat` structure that you want to fill. For more information, see [stat\(\)](#) (p. 3164).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_stat()* function populates the passed *stat* structure based on information from the passed *attr* structure and the context pointer, *ctp*.

This is typically used when the resource manager is handling the `_IO_STAT` message, and needs to format the current status information for the resource.

Returns:

EOK on successful completion, or an [errno](#) (p. 539) value if an error occurred.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_stat_default()

Default handler for _IO_STAT messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_stat_default( resmgr_context_t *ctp,
                        io_stat_t *msg,
                        iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_stat_t](#) (p. 1356) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_stat_default()* function implements POSIX semantics for the client's *stat()* or *fstat()* call, which is received as an `_IO_STAT` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *stat* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The *iofunc_stat_default()* function calls:

- [iofunc_time_update\(\)](#) (p. 1364), to ensure that the time entries in the *ocb->attr* structure are current and valid
- [iofunc_stat\(\)](#) (p. 1353) to construct a status entry based on the information in the *ocb->attr* structure.

io_stat_t structure

The `io_stat_t` structure holds the `_IO_STAT` message received by the resource manager:

```
struct _io_stat {
    uint16_t          type;
    uint16_t          combine_len;
    uint32_t          zero;
};

typedef union {
    struct _io_stat    i;
    struct stat        o;
} io_stat_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The *i* member is a structure of type `_io_stat` that contains the following members:

type

`_IO_STAT.`

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

The *o* member is a structure of type `stat`; for more information, see [stat\(\)](#) (p. 3164).

Returns:

-1, to indicate to the resource manager library that it should return one part from the *ctp->iov* structure (see [resmgr_context_t](#) (p. 2598)) to the client, or an [errno](#) (p. 539) value if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

iofunc_sync()

Indicate if synchronization is needed

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_sync( resmgr_context_t* ctp,
                 iofunc_ocb_t* ocb,
                 int ioflag );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

ioflag

The operation being performed on the OCB:

- `_IO_FLAG_WR` — writing.
- `_IO_FLAG_RD` — reading.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_sync()* function indicates if some form of synchronization is needed.

Returns:

O_DSYNC

Data integrity is needed.

O_SYNC

File integrity is needed.

O

Synchronization isn't needed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_sync_default()

Default handler for _IO_SYNC messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_sync_default( resmgr_context_t *ctp,
                        io_sync_t *msg,
                        iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_sync_t](#) (p. 1362) structure that contains the message that the resource manager received. For more information, see [iofunc_sync_verify\(\)](#).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *iofunc_sync_default()* function implements POSIX semantics for the client's *sync()* call, which is received as an `_IO_SYNC` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *sync* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The *iofunc_sync_default()* function calls *iofunc_sync_verify()* (p. 1362) to see if the client can synchronize the resource.

Returns:**EINVAL**

The resource doesn't support synchronizing.

EOK

The client can synchronize the resource.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_sync_verify()

Verify permissions to sync

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_sync_verify( resmgr_context_t *ctp,
                       io_sync_t *msg,
                       iofunc_ocb_t *ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the `io_sync_t` structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_sync_verify()* function verifies that the client has permission to synchronize.

io_sync_t structure

The `io_sync_t` structure holds the `_IO_SYNC` message received by the resource manager:

```
struct _io_sync {
    uint16_t      type;
    uint16_t      combine_len;
    uint32_t      flag;
};

typedef union {
```

```
    struct _io_sync      i;  
} io_sync_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_sync` that contains the following members:

type

`_IO_SYNC`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

flag

One of:

- `O_DSYNC`
- `O_RSYNC`
- `O_SYNC`

For more information about these flags, see [open\(\)](#) (p. 1905).

Returns:

EINVAL

The resource doesn't support syncing.

EOK

The client has read access to this resource.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_time_update()

Update time stamps

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_time_update( iofunc_attr_t* attr );
```

Arguments:

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_time_update()* function examines the *flags* member in the passed *attr* structure against the bits `IOFUNC_ATTR_ETIME`, `IOFUNC_ATTR_MTIME`, and `IOFUNC_ATTR_CTIME`. If any of these bits are set, the corresponding time member of *attr* (e.g. *attr->etime*) isn't valid. This function updates all invalid *attr* members to the current time.

If *iofunc_time_update()* makes any change to the *attr* structure's time members, it sets `IOFUNC_ATTR_DIRTY_TIME` in the *attr* structure's *flags* member. This function always clears the `IOFUNC_ATTR_ETIME`, `IOFUNC_ATTR_MTIME`, and `IOFUNC_ATTR_CTIME` bits from *attr->flags*.

Returns:

EOK

Successful completion.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_unblock()

Unblock OCBs

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_unblock( resmgr_context_t * ctp,
                   iofunc_attr_t * attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_unblock()* function unblocks any clients that are blocked on any internal resource manager structures.



Currently, this involves only the advisory lock list that's maintained by the attribute.

If a client connection is found:

- that client is unblocked, and is replied to with the error `EINTR`.
- *iofunc_unblock()* returns `_RESMGR_NOREPLY`.

If no client connection is found, *iofunc_unblock()* returns `_RESMGR_DEFAULT`.

Returns:

_RESMGR_DEFAULT

No client connection was found.

_RESMGR_NOREPLY

A client connection has been unblocked.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_unblock_default()

Default unblock handler

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_unblock_default( resmgr_context_t * ctp,
                           io_pulse_t * msg,
                           iofunc_ocb_t * ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_pulse_t](#) structure that describes the pulse that the resource manager received.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_unblock_default()* function calls [iofunc_unblock\(\)](#) (p. 1366).

The *iofunc_unblock_default()* function implements the functionality required when the client requests to be unblocked (e.g. a signal or timeout).



If your resource manager uses [iofunc_lock_default\(\)](#) (p. 1281), you *must* use both *iofunc_unblock_default()* and [iofunc_close_dup_default\(\)](#) (p. 1251) in their respective handlers, as they provide necessary ancillary functionality for

managing file locks. This is because file locks are owned by the process, and aren't inherited by the child process.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *unblock* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The unblock message is synthesized by the resource-manager shared library when a client wishes to unblock from its [MsgSendv\(\)](#) (p. 1801) to the resource manager. The [iofunc_unblock_default\(\)](#) function takes care of freeing up any locks that the client may have placed on the resource.

Returns:

`_RESMGR_DEFAULT`

No client connection was found.

`_RESMGR_NOREPLY`

A client connection has been unblocked.

Examples:

If your lock handler called [iofunc_lock_default\(\)](#) (p. 1281), your unblock handler must call [iofunc_unblock_default\(\)](#):

```
int my_unblock_handler (resmgr_context_t *ctp, io_pulse_t *msg, RESMGR_OCB_T *ocb)
{
    int status;

    if((status = iofunc_unblock_default(...)) != _RESMGR_DEFAULT)
    {
        return status;
    }

    /* Do any additional work required to look for a client to unblock. */
    ...
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_unlink()

Verify that an entry can be unlinked

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_unlink( resmgr_context_t* ctp,
                  io_unlink_t* msg,
                  iofunc_attr_t* attr,
                  iofunc_attr_t* dattr,
                  struct _client_info* info );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_unlink_t](#) (p. 1371) structure that contains the message that the resource manager received; see below.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the resource.

dattr

NULL, or a pointer to the [iofunc_attr_t](#) structure that describes the characteristics of the parent directory.

info

NULL, or a pointer to a [_client_info](#) structure that contains information about the client. For information about this structure, see [ConnectClientInfo\(\)](#) (p. 356).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_unlink()* function verifies that the *msg* specifies valid semantics for an unlink, and that the client is allowed to unlink the resource, as specified by a combination of who the client is (*info*), and the resource attributes *attr*, *dattr*, *attr->uid* and *attr->gid*.

If a directory entry is being removed, *iofunc_unlink()* checks to see that the directory is empty. The *iofunc_unlink()* function also updates the time stamps, and decrements the link count for the entry.

io_unlink_t structure

The *io_unlink_t* structure holds the `_IO_CONNECT` message received by the resource manager:

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_unlink_t;
```

This message structure is a union of an input message (coming to the resource manager), `_io_connect` (p. 1189), and two possible output or reply messages (going back to the client):

- `_io_connect_link_reply` (p. 1196) if the reply is redirecting the client to another resource
- Or:
- `_io_connect_ftype_reply` (p. 1195) if the reply consists of a status and a file type.

Returns:**EOK**

Successful completion.

ENOTDIR

Attempt to unlink a nondirectory entry using directory semantics, (e.g. `rmdir file`).

EINVAL

Attempt to remove the "." directory.

ENOTEMPTY

Attempt to remove a directory that isn't empty.

EROFS

Attempt to remove an entry on a read-only filesystem.

EACCES

The client doesn't have permissions to do the operation.

EPERM

The group ID or owner ID didn't match.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_unlock_ocb_default()

Default handler for the unlock_ocb callout

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_unlock_ocb_default(
    resmgr_context_t * ctp,
    void * reserved,
    iofunc_ocb_t * ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

reserved

This argument must be NULL.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_unlock_ocb_default()* function calls [iofunc_attr_unlock\(\)](#) (p. 1228) to enforce unlocking on the attributes for the group of messages that were sent by the client.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to [resmgr_attach\(\)](#) (p. 2580), at the *unlock_ocb* position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

Returns:

EOK

Success.

EAGAIN

On the first use, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_untime()

Update time stamps

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_untime( resmgr_context_t* ctp,
                  io_untime_t* msg,
                  iofunc_ocb_t* ocb,
                  iofunc_attr_t* attr );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_untime_t](#) (p. 1376) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

attr

A pointer to the [iofunc_attr_t](#) (p. 1221) structure that describes the characteristics of the device that's associated with your resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_untime()* helper function examines the *flags* member in the passed *attr* structure and sets the `IOFUNC_ATTR_ETIME` and `IOFUNC_ATTR_MTIME` bits if requested.

The function sets the `IOFUNC_ATTR_CTIME` and `IOFUNC_ATTR_DIRTY_TIME` bits. It then calls [iofunc_time_update\(\)](#) (p. 1364) to update the file times.

io_untime_t structure

The `io_untime_t` structure holds the `_IO_ETIME` message received by the resource manager:

```
struct _io_untime {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       cur_flag;
    struct utimbuf times;
};

typedef union {
    struct _io_untime i;
} io_untime_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type `_io_untime` that contains the following members:

type

`_IO_ETIME`.

combine_len

If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

cur_flag

If set, `iofunc_untime()` ignores the *times* member, and set the appropriate file times to the current time.

times

A `utimbuf` structure that specifies the time to use when setting the file times. For more information about this structure, see [utime\(\)](#) (p. 3588).

Returns:

EACCES

The client doesn't have permissions to do the operation.

EFAULT

A fault occurred when the kernel tried to access the *info* buffer.

EINVAL

The client process is no longer valid.

ENOSYS

NULL was passed in *info*.

EOK

Successful completion.

EPERM

The group ID or owner ID didn't match.

EROFS

Attempt to remove an entry on a read-only filesystem.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_utime_default()

Default handler for _IO_ETIME messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_utime_default( resmgr_context_t* ctp,
                          io_etime_t* msg,
                          iofunc_ocb_t* ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_etime_t](#) (p. 1376) structure that contains the message that the resource manager received; see *iofunc_etime()*.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_etime_default()* function implements POSIX semantics for the client's *utime()* call, which is received as an `_IO_ETIME` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the *io_funcs* argument to *resmgr_attach()* (p. 2580), at the *utime* position, or you can call *iofunc_func_init()* (p. 1271) to initialize all of the functions to their default values.

The *iofunc_etime_default()* function calls *iofunc_etime()* (p. 1375) to do the actual work. It verifies that the client has the necessary permissions to effect a *utime* on the device.

If so, the utime is performed, modifying elements of the *ocb->attr* structure. This function takes care of updating these bits in the flags member of *ocb->attr*:

- IOFUNC_ATTR_ETIME
- IOFUNC_ATTR_CTIME
- IOFUNC_ATTR_MTIME
- IOFUNC_ATTR_DIRTY_TIME
- IOFUNC_ATTR_DIRTY_MODE

The *iofunc_utime()* function then calls [iofunc_time_update\(\)](#) (p. 1364) to update the appropriate time fields in *ocb->attr*.

Returns:

EOK

Successful completion.

EROFS

An attempt was made to utime a read-only filesystem.

EACCES

The client doesn't have permissions to do the operation.

EPERM

The group ID or owner ID didn't match.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_write_default()

Default handler for _IO_WRITE messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_write_default( resmgr_context_t* ctp,
                        io_write_t* msg,
                        iofunc_ocb_t* ocb );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_write_t](#) (p. 1383) structure that contains the message that the resource manager received. For more information, see [iofunc_write_verify\(\)](#).

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The [iofunc_write_default\(\)](#) function implements POSIX semantics for the client's `write()` call, which is received as an `_IO_WRITE` message by the resource manager.

You can place this function directly into the [resmgr_io_funcs_t](#) (p. 2610) table passed as the `io_funcs` argument to [resmgr_attach\(\)](#) (p. 2580), at the `write` position, or you can call [iofunc_func_init\(\)](#) (p. 1271) to initialize all of the functions to their default values.

The `iofunc_write_default()` function calls [iofunc_write_verify\(\)](#) (p. 1382) to verify the permissions on the file. If the permissions are correct, `iofunc_write_default()` indicates that it wrote zero bytes.

Returns:**EBADF**

The client doesn't have write access to this resource.

EINVAL

An unknown *xtype* was given.

EOK

The client has write access to this resource.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

iofunc_write_verify()

Verify a client's write access to a resource

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_write_verify( resmgr_context_t* ctp,
                        io_write_t* msg,
                        iofunc_ocb_t* ocb,
                        int* nonblock );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

A pointer to the [io_write_t](#) (p. 1383) structure that contains the message that the resource manager received; see below.

ocb

A pointer to the [iofunc_ocb_t](#) (p. 1319) structure for the Open Control Block that was created when the client opened the resource.

nonblock

NULL, or a pointer to a location where the function can store a value that indicates whether or not the device is nonblocking:

- Nonzero — the client doesn't want to be blocked (i.e. `O_NONBLOCK` was set).
- Zero — the client wants to be blocked.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iofunc_write_verify()* function checks that the client that sent the write message actually has write access to the resource, and, optionally (if *nonblock* isn't NULL), sets *nonblock* to *O_NONBLOCK* or 0.

The write permission check is done against *ocb->ioflag*.

Note that the *io_write_t* message has an override flag called *msg->i.xtype*. This flag allows the client to override the default blocking behavior for the resource on a per-request basis. This override flag is checked, and returned in the optional *nonblock*.

In *write* callout for your resource manager, you'll need to indicate how many bytes were written. You can do this with the macro:

```
_IO_SET_WRITE_NBYTES( resmgr_context_t *ctp,  
                      int nbytes )
```

io_write_t structure

The *io_write_t* structure holds the *_IO_WRITE* message received by the resource manager:

```
struct _io_write {  
    uint16_t                                type;  
    uint16_t                                combine_len;  
    int32_t                                 nbytes;  
    uint32_t                                xtype;  
    uint32_t                                zero;  
    /* unsigned char                        data[nbytes]; */  
};  
  
typedef union {  
    struct _io_write                                i;  
    /* nbytes is returned with MsgReply */  
} io_write_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, *i*.

The *i* member is a structure of type *_io_write* that contains the following members:

type

_IO_WRITE.

combine_len

If the message is a combine message, *_IO_COMBINE_FLAG* is set in this member. For more information, see Combine Messages chapter of *Writing a Resource Manager*.

nbytes

The number of bytes that the client wants to write.

xtype

Extended type information; one of:

- `_IO_XTYPE_NONE`
- `_IO_XTYPE_READCOND`
- `_IO_XTYPE_QUEUE`
- `_IO_XTYPE_TCPIP`
- `_IO_XTYPE_TCPIP_MSG`
- `_IO_XTYPE_OFFSET`
- `_IO_XTYPE_REGISTRY`
- `_IO_XFLAG_DIR_EXTRA_HINT` — this flag is valid only when reading from a directory. The filesystem should normally return extra directory information when it's easy to get. If this flag is set, it is a hint to the filesystem to try harder (possibly causing media lookups) to return the extra information. The most common use would be to return `_DTYPE_LSTAT` information.
- `_IO_XFLAG_NONBLOCK`
- `_IO_XFLAG_BLOCK`

For more information, see “Handling other read/write details” in the Handling Read and Write Messages chapter of *Writing a Resource Manager*.

The commented-out declaration for *data* indicates that *nbytes* bytes of data immediately follow the `_io_write` structure.

Returns:

EOK

The client has write access to this resource.

EBADF

The client doesn't have write access to this resource.

EBADMSG

The length of the message is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ionotify()

Arm a resource manager

Synopsis:

```
#include <unistd.h>
#include <sys/iomsg.h>

int ionotify ( int          fd,
              int          action,
              int          flags,
              const struct sigevent *event );
```

Arguments:

fd

The file descriptor associated with the resource manager that you want to notify.

action

The type of arming action to take; see “[Actions](#) (p. 1388),” below.

flags

The types of conditions that can be checked for notification; see “[Flags](#) (p. 1387),” below.

event

A pointer to a [sigevent](#) (p. 2961) structure that defines the event that you want the resource manager to send as a notification, or NULL to disarm a notification.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *ionotify()* function arms the resource manager associated with *fd* to send the event notification *event*. The event is sent when a condition specified by a combination of *action* and *flags* occurs.

Flags

The *flags* argument specifies the types of conditions that can be checked for notification. Each resource manager maintains a different context for each notification condition. Only those notification bits specified are affected. In the following example, the second call to *ionotify()* doesn't affect the first, since it specifies a different notification:

```
ionotify( fd, _NOTIFY_ACTION_POLLARM,
         _NOTIFY_COND_INPUT, &event );
ionotify( fd, _NOTIFY_ACTION_POLLARM,
         _NOTIFY_COND_OUTPUT, &event );
```

The conditions specified by *flags* are:

_NOTIFY_COND_OBAND

Out-of-band data is available. The definition of out-of-band data depends on the resource manager.

_NOTIFY_COND_OUTPUT

There's room in the output buffer for more data. The amount of room available needed to satisfy this condition depends on the resource manager. Some resource managers may default to an empty output buffer, while others may choose some percentage of the buffer empty.

_NOTIFY_COND_INPUT

There's input data available. The amount of data available defaults to 1. For a character device such as a serial port, this would be a character. For a POSIX message queue, it would be a message. Each resource manager selects an appropriate object.

_NOTIFY_COND_EXTEN

The conditions are defined with the following extended flags:

- **_NOTIFY_CONDE_RDNORM** — normal data is available; this is the same as **_NOTIFY_COND_INPUT**.
- **_NOTIFY_CONDE_WRNORM** — room for normal data; this is the same as **_NOTIFY_COND_OUTPUT**.
- **_NOTIFY_CONDE_RDBAND** — out-of-band data is available; this is the same as **_NOTIFY_COND_OBAND**.
- **_NOTIFY_CONDE_PRI** — priority data is available.
- **_NOTIFY_CONDE_WRBAND** — room for OOB data.
- **_NOTIFY_CONDE_ERR** — an error occurred on the device or stream.
- **_NOTIFY_CONDE_HUP** — the device has been disconnected.
- **_NOTIFY_CONDE_NVAL** — the file descriptor is invalid.

The method for changing the default number for `_NOTIFY_COND_OUTPUT` and `_NOTIFY_COND_INPUT` depends on the device. For example, character special devices can call [`readcond\(\)`](#) (p. 2509).

For resource managers that support both an edited and raw mode, the mode should be set to raw to ensure proper operation of `ionotify()`.

The above flags are located in the top bits of `flags`. They are defined by `_NOTIFY_COND_MASK`.

In the case of an asynchronous notification using the passed `event`, such as a QNX Neutrino pulse or queued realtime signal, the 32-bit value in `event->sigev_value.sival_int` is returned to you unmodified, *unless* you've selected the `SI_NOTIFY` code, in which case the top bits (defined by `_NOTIFY_COND_MASK`) are set to the active notifications. In this case, you should limit the `sival_int` to the mask defined by `_NOTIFY_DATA_MASK`.

For example, the Unix `select()` function specifies `SI_NOTIFY` and uses the allowable data bits of `sival_int` as a serial number.



If you're using the `SI_NOTIFY` code, then you should clear the bits as specified by `_NOTIFY_COND_MASK` in the `sigev_value` field — the resource manager only ever ORs in a value, it never clears the bits.

Actions

The `action` argument specifies the type of arming action to take. When a condition is armed, the resource manager monitors it and, when met, delivers `event` using [`MsgDeliverEvent\(\)`](#) (p. 1736). When an event is delivered, it's always disarmed except where noted below.

Note that for transition arming (as specified by an `action` of `_NOTIFY_ACTION_TRANARM`, only *one* notification of that type can be outstanding *per device*. When the transition arm fires, it's removed.

Each action is designed to support a specific notification type as follows:

`_NOTIFY_ACTION_POLL`

This action does a poll of the notification conditions specified by `flags`. It never arms an event, and it cancels all other asynchronous event notifications set up by a previous call to `ionotify()`. This also allows it to be used as a simple “disarm” call.

Returns active conditions as requested by `flags`.

`_NOTIFY_ACTION_POLLARM`

This action does a poll in the same way as `_NOTIFY_ACTION_POLL`.

However, if none of the conditions specified in `flags` are present then each

condition specified in *flags* is armed. If any condition is met, none of the conditions are armed. The Unix *select()* function uses *ionotify()* with this action.

Returns active conditions as requested by *flags*.

`_NOTIFY_ACTION_TRANARM`

This action arms for transitions of the notification conditions specified by *flags*. A transition is defined as a data transition from empty to nonempty on input. Its use on output isn't defined. Note that if there is data available when this call is used, a data transition *won't* occur. To generate an event using this type of notification, you must arm the event and then drain the input using a nonblocking read. After this point, new input data causes the event to be delivered. The *mq_notify()* function uses *ionotify()* with this action.

Since this arms for a transition, the return value is always zero.



There's a limit of one notification per message queue or resource manager. If a notification was already armed for the specified resource, *ionotify()* indicates an error of **EBUSY**.

You can use the `_NOTIFY_ACTION_POLLARM` or `_NOTIFY_ACTION_POLL` action to generate events that are level- as opposed to transition-oriented.

When an action is armed in a resource manager, it remains armed until:

- A thread sets a new action (this disarms any current action and possibly arms a new action),
- The event is delivered and the action wasn't a continuous one,
- The thread closes the device.

Returns:

Active conditions as requested by *flags*. In the case of a transition action, a zero is returned. If an error occurs, -1 is returned (*errno* (p. 539) is set).

Errors:

EBADF

The connection indicated by *fd* doesn't exist, or *fd* is no longer connected to a channel.

EBUSY

A notification was already armed for this resource; this function enforces a restriction of one per message queue or resource manager.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EINTR

The call was interrupted by a signal.

ENOMEM

The resource manager couldn't allocate a notify entry to save the request.

ENOSYS

The requested action isn't supported by this resource manager.

ESRVRFAULT

A fault occurred in a server's address space while accessing the server's message buffers. This may have occurred on the receive or the reply.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

IP

Internet Protocol

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_RAW,
            proto );
```

Description:

IP is the transport layer protocol used by the Internet protocol family. You may set options at the IP level when you're using higher-level protocols based on IP, such as TCP and UDP. You may also access IP through a “raw socket” (when you're developing new protocols or special-purpose applications).

There are several IP-level [setsockopt\(\)](#) (p. 2909) and [getsockopt\(\)](#) (p. 962) options. You can use `IP_OPTIONS` to provide IP options to be transmitted in the IP header of each outgoing packet or to examine the header options on incoming packets. IP options may be used with any socket type in the Internet family. The format of IP options to be sent is that specified by the IP protocol specification (*RFC 791*), with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address is extracted from the option list and the size adjusted accordingly before use. To disable previously specified options, use a zero-length buffer:

```
setsockopt(s, IPPROTO_IP, IP_OPTIONS, NULL, 0);
```

You can use `IP_TOS` and `IP_TTL` to set the type-of-service and time-to-live fields in the IP header for `SOCK_STREAM` and `SOCK_DGRAM` sockets. For example:

```
int tos = IPTOS_LOWDELAY;          /* see <netinet/ip.h> */
setsockopt(s, IPPROTO_IP, IP_TOS, &tos, sizeof(tos));

int ttl = 60;                      /* max = 255 */
setsockopt(s, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
```

If the `IP_RECVSTADDR` option is enabled on a `SOCK_DGRAM` or `SOCK_RAW` socket, the [recvmsg\(\)](#) (p. 2539) call returns the destination IP address for a UDP datagram. The `msg_control` field in the `msghdr` structure points to a buffer that contains a `cmsg_hdr` structure followed by the IP address. The `cmsg_hdr` fields have the following values:

```
cmsg_len = sizeof(struct cmsg_hdr) + sizeof(struct in_addr)
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVSTADDR
```

If the `IP_RECVIF` option is enabled on a `SOCK_DGRAM` or `SOCK_RAW` socket, the `recvmsg()` call returns a `struct sockaddr_dl` corresponding to the interface on which the packet was received. The `msg_control` field in the `msg_hdr` structure points to a buffer that contains a `cmsg_hdr` structure followed by the `struct sockaddr_dl`. The `cmsg_hdr` fields have the following values:

```
cmsg_len = sizeof(struct cmsg_hdr) + sizeof(struct sockaddr_dl)
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVIF
```

Raw IP sockets are connectionless, and are normally used with the [sendto\(\)](#) (p. 2837) and [recvfrom\(\)](#) (p. 2536) calls, although you can also use [connect\(\)](#) (p. 348) to fix the destination for future packets (in which case you can use the [read\(\)](#) (p. 2502) or [recv\(\)](#) (p. 2533) and [write\(\)](#) (p. 3737) or [send\(\)](#) (p. 2829) system calls).

If the `proto` parameter to `socket()` is 0, the default protocol `IPPROTO_RAW` is used for outgoing packets, and only incoming packets destined for that protocol are received. If `proto` is nonzero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with), unless the `IP_HDRINCL` option has been set. Incoming packets are received with IP header and options intact.

`IP_HDRINCL` indicates the complete IP header is included with the data and may be used only with the `SOCK_RAW` type.

```
#include <netinet/ip.h>

int hinc1 = 1; /* 1 = on, 0 = off */
setsockopt(s, IPPROTO_IP, IP_HDRINCL, &hinc1, sizeof(hinc1));
```

The program must set all the fields of the IP header, including the following:

```
ip->ip_v = IPVERSION;
ip->ip_hl = hlen >> 2;
ip->ip_id = 0; /* 0 means kernel set appropriate value */
ip->ip_off = offset;
```

If the header source address is set to `INADDR_ANY`, the kernel chooses an appropriate address.

Multicasting

IP multicasting is supported only on `AF_INET` sockets of type `SOCK_DGRAM` and `SOCK_RAW`, and only on networks where the interface driver supports multicasting.

Multicast Options

IP_MULTICAST_TTL

Change the time-to-live (TTL) for outgoing multicast datagrams in order to control the scope of the multicasts:

```
u_char ttl; /* range: 0 to 255, default = 1 */
```

```
setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, &tll, sizeof(ttl));
```

Datagrams with a TTL of 1 aren't forwarded beyond the local network. Multicast datagrams with a TTL of 0 aren't transmitted on any network, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback hasn't been disabled on the sending socket (see below). Multicast datagrams with TTL greater than 1 may be forwarded to other networks if a multicast router is attached to the local network.

IP_MULTICAST_IF

For hosts with multiple interfaces, each multicast transmission is sent from the primary network interface. The `IP_MULTICAST_IF` option overrides the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

where *addr* is the local IP address of the desired interface or `INADDR_ANY` to specify the default interface. You can get an interface's local IP address and multicast capability by sending the `SIOCGIFCONF` and `SIOCGIFFLAGS` requests to *ioctl()* (p. 1198). Normal applications shouldn't need to use this option.

IP_MULTICAST_LOOP

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. The `IP_MULTICAST_LOOP` option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop; /* 0 = disable, 1 = enable (default) */
setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions. It shouldn't generally be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender doesn't belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

IP_ADD_MEMBERSHIP

A host must become a member of a multicast group before it can receive datagrams sent to the group. To join a multicast group, use the `IP_ADD_MEMBERSHIP` option:

```
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

where *mreq* is the following structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* multicast group to join */
    struct in_addr imr_interface; /* interface to join on */
}
```

Set *imr_interface* to `INADDR_ANY` to choose the default multicast interface, or to the IP address of a particular multicast-capable interface if the host is multihomed. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface. Up to `IP_MAX_MEMBERSHIPS` (currently 20) memberships may be added on a single socket.

IP_DROP_MEMBERSHIP

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where *mreq* contains the same values as used to add the membership. Memberships are dropped when the socket is closed or the process exits.

Based on:

RFC 791

Returns:

A descriptor referencing the socket, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EADDRNOTAVAIL

You tried to create a socket with a network address for which no network interface exists.

EISCONN

You tried to establish a connection on a socket that already has one or to send a datagram with the destination address specified, but the socket is already connected.

ENOBUFS

The system ran out of memory for an internal data structure.

ENOTCONN

You tried to send a datagram, but no destination address was specified and the socket hasn't been connected.

The following error specific to IP may occur when setting or getting IP options:

EINVAL

An unknown socket option name was given. The IP option field was improperly formed — an option field was shorter than the minimum value or longer than the option buffer provided.

IPsec

Internet security protocol

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet6/ipsec.h>

int socket( PF_KEY,
            SOCK_RAW,
            PF_KEY_V2 );
```

Description:

IPsec is a security protocol for the Internet Protocol layer. It consists of these sub-protocols:

AH (Authentication Header)

Guarantees the integrity of the IP packet and protects it from intermediate alteration or impersonation by attaching a cryptographic checksum computed by one-way hash functions.

ESP (Encapsulated Security Payload)

Protects the IP payload from wire-tapping by encrypting it using secret-key cryptography algorithms.

IPsec has these modes of operation:

- Transport — protects peer-to-peer communication between end nodes.
- Tunnel — supports IP-in-IP encapsulation operation and is designed for security gateways, like VPN configurations.

Kernel interface

The IPsec protocol behavior is controlled by these engines:

- Key management engine — accessed from an application using PF_KEY sockets. The *RFC 2367* specification defines the PF_KEY socket API.
- Policy engine — accessed with the PF_KEY API, the [setsockopt\(\)](#) (p. 2909) operations, and the [sysctl\(\)](#) (p. 3311) interface. (The `sysctl` utility is a cover for the [sysctl\(\)](#) (p. 3311) function.) The [setsockopt\(\)](#) (p. 2909) function defines per-socket behavior and the [sysctl\(\)](#) (p. 3311) interface defines host-wide default behavior.

These engines are located in the socket manager. The socket manager implements the PF_KEY interface and allows you to define IPsec policy similar to per-packet filters.

Note that the socket manager code doesn't implement the dynamic encryption key exchange protocol IKE (Internet Key Exchange) — that implementation should be done at the application level (usually as daemons), using the previously described APIs.

Policy management

The socket manager implements experimental policy management. You can manage the IPsec policy in these ways:

- Configure a per-socket policy using [setsockopt\(\)](#) (p. 2909).
- Configure the socket manager packet filter-based policy using the PF_KEY interface or via the `setkey` utility.

In this case, the default policy is allowed with the `setkey`. By configuring the policy to default, you can use the system-wide `sysctl` utility variables. (The `sysctl` utility displays various runtime options.)

If the socket manager finds no matching policy, the system-wide default value is applied.

For a list of `net.inet.ipsec6.*` variables, see the `sysctl` utility in the *Utilities Reference*.

Miscellaneous sysctl variables

The following variables are accessible via the `sysctl` utility for tweaking socket manager IPsec behavior:

Name	Type	Changeable?
<code>net.inet.ipsec.ah_clearatos</code>	Integer	Yes
<code>net.inet.ipsec.ah_offsetmask</code>	Integer	Yes
<code>net.inet.ipsec.dfbit</code>	Integer	Yes
<code>net.inet.ipsec.ecn</code>	Integer	Yes
<code>net.inet.ipsec.debug</code>	Integer	Yes
<code>net.inet6.ipsec6.ecn</code>	Integer	Yes
<code>net.inet6.ipsec6.debug</code>	Integer	Yes

The variables are interpreted as follows:

ipsec.ah_clearatos

When computing AH authentication data, the socket manager clears the type-of-service field in the IPv4 header if the value is set to a nonzero value. The variable tweaks AH behavior to interoperate with devices that implement

RFC 1826 AH. Set this to a nonzero value (clear the type-of-service field) if you want to conform to *RFC 2402*.

ipsec.ah_offsetmask

When computing AH authentication data, the socket manager includes the 16-bit fragment offset field (including flag bits) in the IPv4 header, after computing a logical “AND” with the variable. This variable tweaks the AH behavior to interoperate with devices that implement *RFC 1826* AH. Set this value to zero (clear the fragment offset field during computation) if you want to conform to *RFC 2402*.

ipsec.dfbit

Configures the socket manager behavior for IPv4 IPsec tunnel encapsulation. The variable is supplied to conform to *RFC 2403* Chapter 6.1.

If the value is set to:	Then:
0	The DF bit on the outer IPv4 header is cleared.
1	The outer DF bit on the header is set from the inner DF bit.
2	The DF bit is copied from the inner header to the outer.

ipsec.ecn

If set to nonzero, the IPv4 IPsec tunnel encapsulation/decapsulation behavior supports ECN (Explicit Congestion Notification), as documented in the IETF draft `draft-ietf-ipsec-ecn-02.txt`.

ipsec.debug

If set to nonzero, debug messages are generated to the `syslog`.

Variables under the *net.inet6.ipsec6* tree have meaning similar to their *net.inet.ipsec* counterparts.

Protocols

Because the IPsec protocol works like a plugin to the INET and [INET6](#) (p. 1100) protocols, IPsec supports most of the protocols defined upon those IP-layer protocols. Some of the protocols, like [ICMP](#) (p. 1045) or [ICMP6](#) (p. 1047), may behave differently with IPsec. This is because IPsec can prevent ICMP or ICMP6 routines from looking into the [IP](#) (p. 1391) payload.

Setting the policy

You can set the policy manually by calling `setkey`, or set it permanently in `/etc/inetd.conf`. Valid policy settings include:

for `setkey`:

```
-P direction discard
-P direction ipsec request ...
-P direction none
```

for `/etc/inetd.conf`:

```
direction bypass
direction entrust
direction ipsec request ...
```

where:

direction

The direction in which the policy is applied. It's either `in` or `out`.

bypass

(`/etc/inetd.conf` only) Bypass the IPsec processing and transmit the packet in clear text. This option is for privileged sockets.

discard

(`setkey` only) Discard the packet matching indexes.

entrust

(`/etc/inetd.conf` only) Consult the Security Policy Database (SPD) in the stack. The SPD is set by `setkey` (see the *Utilities Reference*).

ipsec request ...

Put the IPsec operation into the packet. You can specify one or more *request* strings using the following format:

```
protocol/mode/src-dst[/level]
```

For detailed descriptions of the arguments in the *request* string, see [below](#) (p. 1399).

none

(`setkey` only) Don't put the IPsec operation into the packet.

Arguments for *request*

protocol

One of:

- ah — Authentication Header. Guarantees the integrity of the IP packets and protects them from intermediate alteration or impersonation, by attaching cryptographic checksums computed by one-way hash functions.
- esp — Encapsulated Security Payload. Protects the IP payload from wire-tapping by encrypting it with secret key cryptography algorithms.
- ipcomp — IP Payload Compression Protocol.

mode

Security protocol to be used, which is one of:

- transport — Protects peer-to-peer communication between end nodes.
- tunnel — Includes IP-in-IP encapsulation operations and is designed for security gateways, like VPN configurations.

dst, src

The “receiving node” (*dst*) and “sending node” (*src*) endpoint addresses of the Security Association (SA). When the direction specified is *in*, *dst* would represent this node and *src* the other node (peer).

If transport is specified as the *mode*, you can omit these values.

level

One of:

- default — The stack should consult the system default policy that's set by the `sysctl` utility.
- require — An SA is required whenever the kernel deals with the packet.
- use — Use an SA if it's available; otherwise, keep the normal operation.
- unique — (`setkey` only) Similar to *require*, but adds the restriction that the SA for outbound traffic is used only for this policy.

You may need the identifier in order to relate the policy and the SA when you define the SA by manual keying. You can put the decimal number as the identifier after *unique*, such as: `unique: number`

The value of *number* must be between 1 and 32767. If the request string is kept unambiguous, the *level* and slash prior to *level* can be omitted. However, you should specify them explicitly to avoid unintended behaviors.



If the *level* isn't specified in the `setkey` command, `unique` is used by default.

Based on:

RFC 2367, RFC 1826, RFC 2402, RFC 2403

Detailed documentation about the IP security protocol may be found at the IPsec FAQ website at <http://www.netbsd.org/Documentation/network/ipsec/>.

Caveats:

The IPsec support is subject to change as the IPsec protocols develop.

There's no single standard for policy engine API, so the policy engine API described herein is just for KAME implementation.

The AH tunnel may not work as you might expect. If you configure the require policy against AH tunnel for inbound, tunneled packets will be rejected. This is because AH authenticates the encapsulating (outer) packet, not the encapsulated (inner) packet.

Under certain conditions, a truncated result may be returned from the socket manager from `SADB_DUMP` and `SADB_SPDDUMP` operations on a `PF_KEY` socket. This occurs if there are too many database entries in the socket manager and the socket buffer for the `PF_KEY` socket is too small. If you manipulate many IPsec key/policy database entries, increase the size of socket buffer.

ipsec_dump_policy()

Generate a readable string from an IPsec policy specification

Synopsis:

```
#include <netinet6/ipsec.h>

char* ipsec_dump_policy(char *buf,
                       char *delim);
```

Arguments:

buf

A pointer to an IPsec policy structure `struct sadb_x_policy`.

delim

Delimiter string, usually a NULL which indicates a space (" ").

Library:

`libipsec`

Use the `-l ipsec` option to `qcc` to link against this library.

Description:

The function *ipsec_dump_policy()* generates a readable string from an IPSEC policy specification. Please refer to [ipsec_set_policy\(\)](#) (p. 1406) for details about the policies.

The *ipsec_dump_policy()* function converts IPsec policy structure into a readable form. Therefore, *ipsec_dump_policy()* is the inverse of *ipsec_set_policy()*. If you set *delim* to NULL, a single whitespace is assumed. The function *ipsec_dump_policy()* returns a pointer to a dynamically allocated string. It is the caller's responsibility to reclaim the region, by using *free()*.

Returns:

A pointer to dynamically allocated string, or NULL if an error occurs.

Examples:

See [ipsec_set_policy\(\)](#) (p. 1406).

Classification:*Unix* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ipsec_get_policylen()

Get the length of the IPsec policy

Synopsis:

```
#include <netinet6/ipsec.h>

int ipsec_get_policylen(char *buf);
```

Arguments:

buf

A pointer to an IPsec policy structure `struct sadb_x_policy`.

Library:

`libipsec`

Use the `-l ipsec` option to `qcc` to link against this library.

Description:

The function *ipsec_get_policylen()* gets the length of the IPsec policy. Please refer to [ipsec_set_policy\(\)](#) (p. 1406) for details about the policies.

You may want the length of the generated buffer when calling *setsockopt()*. The function *ipsec_get_policylen()* returns the length.

Returns:

The size of the buffer, or a negative value if an error occurs.

Examples:

See [ipsec_set_policy\(\)](#) (p. 1406).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

ipsec_set_policy()

Generate an IPsec policy specification structure from a readable string

Synopsis:

```
#include <netinet6/ipsec.h>

char* ipsec_set_policy( char *policy,
                       int len );
```

Arguments:

policy

A string that describes a struct `sadb_x_policy` and optionally a struct `sadb_x_ipsecrequest`, formatted as described below.

len

The length of the policy string.

Library:

libipsec

Use the `-l ipsec` option to `qcc` to link against this library.

Description:

The *ipsec_set_policy()* function generates an IPsec policy specification structure, namely a struct `sadb_x_policy` and potentially a struct `sadb_x_ipsecrequest` from a policy specification given as a C string. The *ipsec_set_policy()* function allocates a buffer, stores the corresponding IPsec policy specification structure in it, and returns a pointer to the buffer.



You should release the buffer returned by *ipsec_set_policy()* by calling [free\(\)](#) (p. 717). See the example below.

The policy specifications can include the following variables:

direction

The direction in which to apply the policy. This must be `in` or `out`

priority_specification

The priority for the policy. This is a signed integer that controls the placement of the policy within the security policy database (SPD). Policies with higher priorities are placed closer to the beginning of the list, and those with lower priorities are placed closer to the end of the list. Policies with equal priorities are added at the end of the group of such policies. You can specify the priority only when `libipsec` has been compiled.

The policy specifications include the following:

direction [priority_specification] discard

Drop any packets that match the policy.

direction [priority_specification] entrust

Consult the SPD defined by `setkey`.

direction [priority_specification] bypass

Bypass the IPsec processing, i.e. transmit packets in clear. This is for privileged sockets.

direction [priority_specification] ipsec request ...

The matching packets are subject to IPsec processing. The `ipsec` string is followed by one or more request strings, which are formatted as below:

protocol / mode / src - dst [/level]

The elements of the string are as follows:

protocol

Either `ah`, `esp`, or `ipcomp`.

mode

Either `transport` or `tunnel`.

src and dst

The IPsec endpoints; *src* is the sending node and *dst* is the receiving node. Therefore, when the direction is `in`, *dst* is this node and *src* is the other node (peer). If the *mode* is `transport`, you can omit both *src* and *dst*.

level

Either `default`, `use`, `require` or `unique`.

- `default` — `io-pkt` should consult the system default policy defined by `sysctl()` (p. 3311).
- `use` — a relevant SA (security association) is used when available, since `io-pkt` may perform IPsec operation against packets when possible. In this case, packets are transmitted in clear (when SA isn't available), or encrypted (when SA is available).
- `require` — a relevant SA is required, since `io-pkt` must perform IPsec operation against packets.
- `unique` is similar to `require`, but it adds the restriction that the SA for outbound traffic be used only for this policy. You may need the identifier in order to relate the policy and the SA when you define the SA by manual keying. You put the decimal number as the identifier like:

`unique: number`

where *number* must be between 1 and 32767.

If the request string is nonambiguous, you can omit the *level* and the slash (/) before it. However, it's best specify the level explicitly, in order to avoid unintended behavior.

If you omit the *level*, the default (which you can use `sysctl` (p. 3311) to get or set) is used. For example:

- If `sysctl` shows `net.inet.ipsec.esp_net_deflev = 1`, then by default, the IPsec policy is required for tunnelling mode. That is, if you define a rule specifying IPsec tunnelling from A to B, then the level is `require`.
- If you use `sysctl` to disable `net.inet.ipsec.esp_net_deflev`, then if you define a rule specifying IPsec tunnelling from A to B, the IPsec rule isn't applied from A to B.

Here's an example of policy information:

```
in discard
out ipsec esp/transport//require
in ipsec ah/transport//require
out ipsec esp/tunnel/10.1.1.2-10.1.1.1/use
in ipsec ipcom/transport//use esp/transport//use
```



This differs from the specification of `setkey`, which doesn't use `entrust` or `bypass`.

Returns:

A pointer to the allocated buffer for policy specification, or NULL if an error occurred.

Examples:

```
#include <netinet6/ipsec.h>
#include <sys/socket.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>

int
main(void)
{
    char *sadb;
    char *policy = "in discard";
    int len;

    sadb = ipsec_set_policy(policy, strlen(policy));

    if (sadb == NULL) {
        fprintf(stderr, "ipsec_set_policy: %s\n", ipsec_strerror());
        return 1;
    }

    len = ipsec_get_policylen(sadb);
    printf("len: %d\n", len);

    policy = NULL;
    policy = ipsec_dump_policy(sadb, NULL);

    if (policy == NULL) {
        fprintf(stderr, "ipsec_dump_policy: %s\n", ipsec_strerror());
        return 1;
    }

    printf("policy: %s\n", policy);

    free(policy);
    free(sadb);

    return 0;
}
```

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ipsec_strerror()

Error messages for IPsec policy manipulation library

Synopsis:

```
#include <netinet6/ipsec.h>

const char *ipsec_strerror( void );
```

Library:

libipsec
Use the -l ipsec option to gcc to link against this library.

Description:

The *ipsec_strerror()* function obtains the error message for the last IPsec call that failed. If you program includes the declaration:

```
extern int ipsec_errcode;
```

you can access the error code from the IPsec policy manipulation library. You can use *ipsec_strerror()* to obtain the error message corresponding to the error code.

Returns:

A pointer to a C string.



Don't modify the string that this function returns. Since *ipsec_strerror()* uses [strerror\(\)](#) (p. 3198) as its underlying function, calling *strerror()* after calling *ipsec_strerror()* overwrites or invalidates the return value from *ipsec_strerror()*.

Examples:

See the example of [ipsec_set_policy\(\)](#) (p. 1406).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	No

IPv6

Internet Protocol version 6

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET6,
            SOCK_RAW,
            proto);
```

Description:

The IP6 protocol is the network-layer protocol used by the Internet Protocol version 6 family (AF_INET6). Options may be set at the IP6 level when using higher-level protocols based on IP6 (such as [TCP](#) (p. 3359) and [UDP](#) (p. 3553)). It may also be accessed through a “raw socket” when developing new protocols, or special-purpose applications.

There are several IP6-level [setsockopt\(\)](#) (p. 2909)/[getsockopt\(\)](#) (p. 962) options. They are separated into the basic IP6 sockets API (defined in *RFC 2553*), and the advanced API (defined in *RFC 2292*). The basic API looks very similar to the API presented in [IP](#) (p. 1391). The advanced API uses ancillary data and can handle more complex cases.



Specifying some of the socket options requires **root** privileges.

Basic IP6 sockets API

You can use the IPV6_UNICAST_HOPS option to set the hoplimit field in the IP6 header on unicast packets. If you specify -1, the socket manager uses the default value. If you specify a value of 0 to 255, the packet uses the specified value as its hoplimit. Other values are considered invalid and result in an error code of EINVAL. For example:

```
int hlim = 60; /* max = 255 */
setsockopt( s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
            &hlim, sizeof(hlim) );
```

The IP6 multicasting is supported only on AF_INET6 sockets of type SOCK_DGRAM and SOCK_RAW, and only on networks where the interface driver supports multicasting.

The IPV6_MULTICAST_HOPS option changes the hoplimit for outgoing multicast datagrams in order to control the scope of the multicasts:

```
unsigned int hlim; /* range: 0 to 255, default = 1 */
setsockopt( s, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
            &hlim, sizeof(hlim) );
```


Datagrams with a hoplimit of 1 aren't forwarded beyond the local network. Multicast datagrams with a hoplimit of 0 won't be transmitted on any network, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback hasn't been disabled on the sending socket (see below). Multicast datagrams with a hoplimit greater than 1 may be forwarded to other networks if a multicast router is attached to the local network.

For hosts with multiple interfaces, each multicast transmission is sent from the primary network interface. The `IPV6_MULTICAST_IF` option overrides the default for subsequent transmissions from a given socket:

```
unsigned int outif;
outif = if_nametoindex("ne0");
setsockopt( s, IPPROTO_IPV6, IPV6_MULTICAST_IF,
            &outif, sizeof(outif) );
```

(The *outif* argument is an interface index of the desired interface, or 0 to specify the default interface.)

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP6 layer for local delivery. The `IPV6_MULTICAST_LOOP` option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop; /* 0 = disable, 1 = enable (default) */
setsockopt( s, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
            &loop, sizeof(loop));
```

This option improves performance for applications that may have no more than one instance on a single host (such as a router daemon), by eliminating the overhead of receiving their own transmissions. Don't use the `IPV6_MULTICAST_LOOP` option if there might be more than one instance of your application on a single host (e.g. a conferencing program), or if the sender doesn't belong to the destination group (e.g. a time-querying program).

A multicast datagram sent with an initial hoplimit greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such a delivery.

A host must become a member of a multicast group before it can receive datagrams sent to the group. To join a multicast group, use the `IPV6_JOIN_GROUP` option:

```
struct ipv6_mreq mreq6;
setsockopt( s, IPPROTO_IPV6, IPV6_JOIN_GROUP,
            &mreq6, sizeof(mreq6) );
```

Note that the *mreq6* argument has the following structure:

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;
    unsigned int ipv6mr_interface;
};
```

Set the *ipv6mr_interface* member to 0 to choose the default multicast interface, or set it to the interface index of a particular multicast-capable interface if the host is multihomed. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface.

To drop a membership, use:

```
struct ipv6_mreq mreq6;
setsockopt( s, IPPROTO_IPV6, IPV6_LEAVE_GROUP,
            &mreq6, sizeof(mreq6) );
```

The *mreq6* argument contains the same values as used to add the membership. Memberships are dropped when the socket is closed or the process exits.

The `IPV6_PORTRANGE` option controls how ephemeral ports are allocated for `SOCK_STREAM` and `SOCK_DGRAM` sockets. For example:

```
int range = IPV6_PORTRANGE_LOW; /* see <netinet/in.h> */
setsockopt( s, IPPROTO_IPV6, IPV6_PORTRANGE, &range,
            sizeof(range) );
```

The `IPV6_BINDV6ONLY` option controls the behavior of the `AF_INET6` wildcard listening socket. The following example sets the option to 1:

```
int on = 1;
setsockopt( s, IPPROTO_IPV6, IPV6_BINDV6ONLY,
            &on, sizeof(on) );
```

If you set the `IPV6_BINDV6ONLY` option to 1, the `AF_INET6` wildcard listening socket accepts IP6 traffic only. If set to 0, the socket accepts IPv4 traffic as well, as if it were from an IPv4 mapped address, such as `::ffff:10.1.1.1`. Note that if you set the option to 0, IPv4 access control gets much more complicated. For example, even if you have no listening `AF_INET` socket on port X, you'll end up accepting IPv4 traffic by an `AF_INET6` listening socket on the same port. The default value for this flag is copied at socket-instantiation time, from the *net.inet6.ip6.bindv6only* variable from the `sysctl` utility. The option affects [TCP](#) (p. 3359) and [UDP](#) (p. 3553) sockets only.

Advanced IP6 sockets API

The advanced IP6 sockets API lets applications specify or obtain details about the IP6 header and extension headers on packets. The advanced API uses ancillary data for passing data to or from the socket manager.

There are also [setsockopt\(\)](#) (p. 2909) / [getsockopt\(\)](#) (p. 962) options to get optional information on incoming packets:

- `IPV6_PKTINFO`
- `IPV6_HOPLIMIT`
- `IPV6_HOPOPTS`
- `IPV6_DSTOPTS`
- `IPV6_RTHDR`

```
int on = 1;
```

```

setsockopt( fd, IPPROTO_IPV6, IPV6_PKTINFO,
            &on, sizeof(on) );
setsockopt( fd, IPPROTO_IPV6, IPV6_HOPLIMIT,
            &on, sizeof(on) );
setsockopt( fd, IPPROTO_IPV6, IPV6_HOPOPTS,
            &on, sizeof(on) );
setsockopt( fd, IPPROTO_IPV6, IPV6_DSTOPTS,
            &on, sizeof(on) );
setsockopt( fd, IPPROTO_IPV6, IPV6_RTHDR,
            &on, sizeof(on) );

```

When any of these options are enabled, the corresponding data is returned as control information by [recvmsg\(\)](#) (p. 2539), as one or more ancillary data objects.

If `IPV6_PKTINFO` is enabled, the destination IP6 address and the arriving interface index are available via struct `in6_pktinfo` on an ancillary data stream. You can pick the structure by checking for an ancillary data item by setting the *cmsg_level* argument to `IPPROTO_IPV6` and the *cmsg_type* argument to `IPV6_PKTINFO`.

If `IPV6_HOPLIMIT` is enabled, the hoplimit value on the packet is made available to the application. The ancillary data stream contains an integer data item with a *cmsg_level* of `IPPROTO_IPV6` and a *cmsg_type* of `IPV6_HOPLIMIT`.

The *inet6_option_space()* family of functions help you parse ancillary data items for `IPV6_HOPOPTS` and `IPV6_DSTOPTS`. Similarly, the *inet6_rthdr_space()* family of functions help you parse ancillary data items for `IPV6_RTHDR`.



The `IPV6_HOPOPTS` and `IPV6_DSTOPTS` values may appear multiple times on an ancillary data stream (note that the behavior is slightly different from the specification). Other ancillary data items appear no more than once.

You can pass ancillary data items with normal payload data, using the [sendmsg\(\)](#) (p. 2833) function. Ancillary data items are parsed by the socket manager, and are used to construct the IP6 header and extension headers. For the *cmsg_level* values listed above, the ancillary data format is the same as the inbound case.

Additionally, you can specify a `IPV6_NEXTHOP` data object. The `IPV6_NEXTHOP` ancillary data object specifies the next hop for the datagram as a socket address structure. In the `cmsg_hdr` structure containing this ancillary data, the *cmsg_level* argument is `IPPROTO_IPV6`, the *cmsg_type* argument is `IPV6_NEXTHOP`, and the first byte of *cmsg_data* is the first byte of the socket address structure.

If the socket address structure contains an IP6 address (e.g. the *sin6_family* argument is `AF_INET6`), then the node identified by that address must be a neighbor of the sending host. If that address equals the destination IP6 address of the datagram, then this is equivalent to the existing `SO_DONTROUTE` socket option.

For applications that don't, or can't use the [sendmsg\(\)](#) (p. 2833) or the [recvmsg\(\)](#) (p. 2539) function, the `IPV6_PKTOPTIONS` socket option is defined. Setting the socket option specifies any of the optional output fields:

```

setsockopt( fd, IPPROTO_IPV6, IPV6_PKTOPTIONS,
            &buf, len );

```

The *buf* argument points to a buffer containing one or more ancillary data objects; the *len* argument is the total length of all these objects. The application fills in this buffer exactly as if the buffer were being passed to the [sendmsg\(\)](#) (p. 2833) function as control information.

The options set by calling [setsockopt\(\)](#) (p. 2909) for `IPV6_PKTOPTIONS` are called “sticky” options because once set, they apply to all packets sent on that socket. The application can call [setsockopt\(\)](#) again to change all the sticky options, or it can call [setsockopt\(\)](#) with a length of 0 to remove all the sticky options for the socket.

The corresponding receive option:

```
getsockopt( fd, IPPROTO_IPV6, IPV6_PKTOPTIONS,  
            &buf, &len );
```

returns a buffer with one or more ancillary data objects for all the optional receive information that the application has previously specified that it wants to receive. The *buf* argument points to the buffer that the call fills in. The *len* argument is a pointer to a value-result integer; when the function is called, the integer specifies the size of the buffer pointed to by *buf*, and on return this integer contains the actual number of bytes that were stored in the buffer. The application processes this buffer exactly as if it were returned by [recvmsg\(\)](#) (p. 2539) as control information.

Advanced API and TCP sockets

When using [getsockopt\(\)](#) (p. 962) with the `IPV6_PKTOPTIONS` option and a [TCP](#) (p. 3359) socket, only the options from the most recently received segment are retained and returned to the caller, and only after the socket option has been set. The application isn't allowed to specify ancillary data in a call to [sendmsg\(\)](#) (p. 2833) on a TCP socket, and none of the ancillary data described above is ever returned as control information by [recvmsg\(\)](#) (p. 2539) on a TCP socket.

Conflict resolution

In some cases, there are multiple APIs defined for manipulating an IP6 header field. A good example is the outgoing interface for multicast datagrams: it can be manipulated by `IPV6_MULTICAST_IF` in the basic API, by `IPV6_PKTINFO` in the advanced API, and by the *sin6_scope_id* field of the socket address structure passed to the [sendto\(\)](#) (p. 2837) function.

In QNX Neutrino, when conflicting options are given to the socket manager, the socket manager gets the value in the following order:

1. options specified by using ancillary data
2. options specified by a sticky option of the advanced API
3. options specified by using the basic API
4. options specified by a socket address.



The conflict resolution is undefined in the API specification and depends on the implementation.

Raw IP6 Sockets

Raw IP6 sockets are connectionless, and are normally used with [sendto\(\)](#) (p. 2837) and [recvfrom\(\)](#) (p. 2536), although you can also use [connect\(\)](#) (p. 348) to fix the destination for future packets (in which case you can use [read\(\)](#) (p. 2502) or [recv\(\)](#) (p. 2533), and [write\(\)](#) (p. 3737) or [send\(\)](#) (p. 2829)).

If *proto* is 0, the default protocol `IPPROTO_RAW` is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is nonzero, that protocol number is used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP6 header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received without the IP6 header or extension headers.

All data sent via raw sockets *must* be in network byte order; all data received via raw sockets is in network-byte order. This differs from the IPv4 raw sockets, which didn't specify a byte ordering and typically used the host's byte order.

Another difference from IPv4 raw sockets is that complete packets (i.e. IP6 packets with extension headers) can't be read or written using the IP6 raw sockets API. Instead, ancillary data objects are used to transfer the extension headers, as described above.

All fields in the IP6 header that an application might want to change (i.e. everything other than the version number) can be modified using ancillary data and/or socket options by the application for output. All fields in a received IP6 header (other than the version number and Next Header fields) and all extension headers are also made available to the application as ancillary data on input. Hence, there's no need for a socket option similar to the IPv4 `IP_HDRINCL` socket option.

When writing to a raw socket, the socket manager automatically fragments the packet if the size exceeds the path MTU, inserting the required fragmentation headers. On input, the socket manager reassembles received fragments, so the reader of a raw socket never sees any fragment headers.

Most IPv4 implementations give special treatment to a raw socket created with a third argument to [socket\(\)](#) (p. 3075) of `IPPROTO_RAW`, whose value is normally 255. We note that this value has no special meaning to an IP6 raw socket (and the IANA currently reserves the value of 255 when used as a next-header field).

For [ICMP6](#) (p. 1047) raw sockets, the socket manager calculates and inserts the mandatory ICMP6 checksum.

For other raw IP6 sockets (i.e. for raw IP6 sockets created with a third argument other than `IPPROTO_ICMPV6`), the application must:

1. Set the new `IPV6_CHECKSUM` socket option to have the socket manager compute and store a pseudo header checksum for output.
2. Verify the received pseudo header checksum on input, discarding the packet if the checksum is in error.

This option prevents applications from having to perform source-address selection on the packets they send. The checksum incorporates the IP6 pseudo-header, defined in Section 8.1 of *RFC 2460*. This new socket option also specifies an integer offset into the user data of where the checksum is located.

```
int offset = 2;
setsockopt( fd, IPPROTO_IPV6, IPV6_CHECKSUM,
            &offset, sizeof(offset));
```

By default, this socket option is disabled. Setting the offset to -1 also disables the option. Disabled means:

1. The socket manager won't calculate and store a checksum for outgoing packets.
2. The socket manager kernel won't verify a checksum for received packets.



- Since the checksum is always calculated by the socket manager for an *ICMP6* (p. 1047) socket, applications can't generate ICMPv6 packets with incorrect checksums (presumably for testing purposes) using this API.
- The `IPV6_NEXTHOP` object/option isn't fully implemented.

Based on:

RFC 2553, RFC 2292, RFC 2460

iruserok(), iruserok_sa()

Check the identity of a remote host

Synopsis:

```
#include <unistd.h>

int iruserok( uint32_t raddr,
              int superuser,
              char * ruser,
              char * luser );

int iruserok_sa( const void * raddr,
                 int superuser,
                 int rlen,
                 char * ruser,
                 char * luser );
```

Arguments:***raddr***

The IP address of the remote host. For *iruserok_sa()*, this argument is specified as `void *` to avoid dependencies between `<unistd.h>` and `<sys/socket.h>`.

rlen

(*iruserok_sa()* only) The length of the IP address that *raddr* points to.

superuser

Nonzero if the local user is the superuser, zero otherwise.

ruser

The name of the remote user.

luser

The name of the local user.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *iruserok()* routine checks the identity of a remote host. It's used by servers to authenticate clients requesting service with *rcmd()* (p. 2491).

The *rcmd()*, *rresvport()* (p. 2670), and *ruserok()* (p. 2696) functions are used by the *rshd* server (see the *Utilities Reference*), among others.

The *iruserok()* function takes a remote host's IP address, two user names, and a flag indicating whether the local user's name is that of the superuser. Then, if the user *isn't* the superuser, it checks the file */etc/hosts.equiv* (described in the *Utilities Reference*).

If that lookup isn't done, or is unsuccessful, the *.rhosts* file in the local user's home directory is checked to see if the request for service is allowed. If this file is owned by anyone other than the user or the superuser, or if it's writable by anyone other than the owner, the check automatically fails.

The *iruserok()* function can handle only IPv4 addresses; *iruserok_sa()* and *ruserok()* can handle other address families as well.

Returns:

0

The machine name is listed in the *hosts.equiv* file, or the host and remote username were found in the *.rhosts* file.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

The error code *EAGAIN* is overloaded to mean “All network ports in use.”

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

isalnum()

Test a character to see if it's alphanumeric

Synopsis:

```
#include <ctype.h>

int isalnum( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *isalnum()* function tests if the argument *c* is an alphanumeric character (a to z, A to Z, or 0 to 9). An alphanumeric character is any character for which [isalpha\(\)](#) (p. 1423) or [isdigit\(\)](#) (p. 1433) is true.

Returns:

Nonzero if *c* is a letter or decimal digit; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
{
    if( isalnum( getchar() ) ) {
        printf( "That's alpha-numeric!\n" );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for char arguments and EOF.

isalpha()

Test a character to see if it's alphabetic

Synopsis:

```
#include <ctype.h>

int isalpha( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isalpha()* function tests if the argument *c* is an alphabetic character (a to z and A to Z). An alphabetic character is any character for which [isupper\(\)](#) (p. 1453) or [islower\(\)](#) (p. 1443) is true.

Returns:

Nonzero if *c* is an alphabetic character; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
{
    if( isalpha( getchar() ) ) {
        printf( "That's alphabetic\n" );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isascii()

Test a character to see if it's a 7-bit ASCII character

Synopsis:

```
#include <ctype.h>

int isascii( int c );
```

Arguments:

c

The character you want to test.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *isascii()* function tests for an ASCII character (in the range 0 to 127).

Returns:

Nonzero if *c* is an ASCII character; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x80, 'Z' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isascii( the_chars[i] ) ) {
            printf( "Char %c is an ASCII character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not an ASCII character\n",
                    the_chars[i] );
        }
    }
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is an ASCII character  
Char  is not an ASCII character  
Char Z is an ASCII character
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isatty()

Test to see if a file descriptor is associated with a terminal

Synopsis:

```
#include <unistd.h>

int isatty( int fildes );
```

Arguments:

fildes

The file descriptor that you want to test.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isatty()* function allows the calling process to determine if the file descriptor *fildes* is associated with a terminal.

Returns:

0

The *fildes* file descriptor doesn't refer to a terminal.

1

The *fildes* file descriptor refers to a terminal.

Examples:

```
/*
 * The following program exits with a status of
 * EXIT_SUCCESS if stderr is a tty; otherwise,
 * EXIT_FAILURE
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    return( isatty( 3 ) ? EXIT_SUCCESS : EXIT_FAILURE );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

isblank()

Test a character to see if it's a blank character

Synopsis:

```
#include <ctype.h>

int isblank( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *isblank()* function tests the given character to see if it is a blank character. In the C locale, the set of blank characters consists of the space (' ') and the horizontal tab ('\t'). The [isspace\(\)](#) (p. 1451) function tests for a wider set of characters.

Returns:

Nonzero if *c* is a blank character; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d, '\n' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isblank( the_chars[i] ) ) {
            printf( "Char %c is a blank character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a blank character\n",
                    the_chars[i] );
        }
    }
}
```

```
        the_chars[i] );
    }
}
return EXIT_SUCCESS;
}
```

This program produces the output:

```
Char A is not a blank character
Char  is a blank character
Char  is a blank character
Char } is not a blank character
Char
is not a blank character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isctrnl()

Test a character to see if it's a control character

Synopsis:

```
#include <ctype.h>

int isctrnl( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isctrnl()* function tests for any control character. An ASCII control character is any character whose value is between 0 and 31.

Returns:

Nonzero if *c* is a control character; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, 'Z' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isctrnl( the_chars[i] ) ) {
            printf( "Char %c is a Control character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a Control character\n",
                    the_chars[i] );
        }
    }
}
```

```
    }  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is not a Control character  
Char  is a Control character  
Char Z is not a Control character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isdigit()

Test a character to see if it's a decimal digit

Synopsis:

```
#include <ctype.h>

int isdigit( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isdigit()* function tests for a decimal digit (characters 0 through 9).

Returns:

Nonzero if *c* is a decimal digit; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', '5', '$' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isdigit( the_chars[i] ) ) {
            printf( "Char %c is a digit character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a digit character\n",
                    the_chars[i] );
        }
    }
}
```

```
        return EXIT_SUCCESS;  
    }
```

produces the output:

```
Char A is not a digit character  
Char 5 is a digit character  
Char $ is not a digit character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isfdtype()

Determine whether a file descriptor refers to a socket

Synopsis:

```
#include <sys/stat.h>

int isfdtype( int filedes,
              int fdtype );
```

Arguments:

filedes

The file descriptor that you want to test.

fdtype

The properties you want to test for. The valid values for *fdtype* include:

- S_IFSOCK — test whether *filedes* is a socket.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *isfdtype()* function determines whether the file descriptor *filedes* has the properties identified by *fdtype*.

This function is based on a POSIX draft; for better portability, call [fstat\(\)](#) (p. 779) and check the buffer that it fills in:



```
if ((buf.st_mode & S_IFMT) == fdtype)
    /* The file descriptor matches fdtype. */
else
    /* The file descriptor doesn't match fdtype. */
```

instead of calling *isfdtype()*.

Returns:

1

The *filedes* file descriptor matches *fdtype*.

0

The *filedes* file descriptor doesn't match *fdtype*.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid file descriptor *filedes*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

isfinite()

Determine if a number is finite

Synopsis:

```
#include <math.h>

int isfinite ( double x );
```

Arguments:

x

The number you want to test.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *isfinite()* macro determines if *x* is finite.

Returns:**True (1)**

The value of *x* is finite.

False (1)

The value of *x* is infinity or NAN.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;

    a = 2;
    b = -0.5;
    c = NAN;
    fp_exception_mask(_FP_EXC_DIVZERO, 1);
    d = 1.0/0.0;
    printf("%f is %s \n", a, (isfinite(a)) ? "finite" : "not finite");
```

```
printf("%f is %s \n", b, (isfinite(b)) ? "finite" : "not finite");
printf("%f is %s \n", c, (isfinite(c)) ? "finite" : "not finite");
printf("%f is %s \n", d, (isfinite(d)) ? "finite" : "not finite");

return(0);
}
```

produces the output:

```
2.000000 is finite
-0.500000 is finite
NAN is not finite
Inf is not finite
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

isgraph()

Test a character to see if it's any printable character except a space

Synopsis:

```
#include <ctype.h>

int isgraph( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isgraph()* function tests for any printable character except a space (' '). The [isprint\(\)](#) (p. 1447) function is similar, except that the space character is also included.

Returns:

Nonzero if *c* is a printable character (except a space); otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isgraph( the_chars[i] ) ) {
            printf( "Char %c is a printable character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a printable character\n",
                    the_chars[i] );
        }
    }
}
```

```
    }  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is a printable character  
Char  is not a printable character  
Char  is not a printable character  
Char } is a printable character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isinf()

Test for infinity

Synopsis:

```
#include <math.h>

#define isinf ( x ) ...
```

Arguments:

x

The number that you want to test.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *isinf()* macro tests to see if a number is “infinity.”

Returns:

1

The value of *x* is infinity.

1

The value of *x* isn't infinity.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;

    a = 2;
    b = -0.5;
    c = NAN;
    fp_exception_mask(_FP_EXC_DIVZERO, 1);
    d = 1.0/0.0;
    printf("%f is %s \n", a,
```

```
        (isinf(a)) ? "infinite" : "not infinite");
    printf("%f is %s \n", b,
        (isinf(b)) ? "infinite" : "not infinite");
    printf("%f is %s \n", c,
        (isinf(c)) ? "infinite" : "not infinite");
    printf("%f is %s \n", d,
        (isinf(d)) ? "infinite" : "not infinite");

    return(0);
}
```

produces the output:

```
2.000000 is not infinite
-0.500000 is not infinite
NAN is not infinite
Inf is infinite
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

islower()

Test a character to see if it's a lowercase letter

Synopsis:

```
#include <ctype.h>

int islower( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *islower()* function tests for any lowercase letter a through z.

Returns:

Nonzero if *c* is a lowercase letter; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 'a', 'z', 'Z' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( islower( the_chars[i] ) ) {
            printf( "Char %c is a lowercase character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a lowercase character\n",
                    the_chars[i] );
        }
    }
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is not a lowercase character  
Char a is a lowercase character  
Char z is a lowercase character  
Char Z is not a lowercase character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isnan()

Test for not-a-number (NaN)

Synopsis:

```
#include <math.h>

#define isnan ( x ) ...
```

Arguments:

x

The number you want to test.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *isnan()* macro determines if *x* is Not-A-Number (NaN).

Returns:

1

The value of *x* is NaN.

1

The value of *x* is a number.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;

    a = 2;
    b = -0.5;
    c = NAN;
    fp_exception_mask(_FP_EXC_DIVZERO, 1);
    d = 1.0/0.0;
```

```
printf("%f is %s \n", a,  
      (isnan(a)) ? "not a number" : "a number");  
printf("%f is %s \n", b,  
      (isnan(b)) ? "not a number" : "a number");  
printf("%f is %s \n", c,  
      (isnan(c)) ? "not a number" : "a number");  
printf("%f is %s \n", d,  
      (isnan(d)) ? "not a number" : "a number");  
return EXIT_SUCCESS;  
}
```

produces the output:

```
2.000000 is a number  
-0.500000 is a number  
NAN is not a number  
Inf is a number
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

isprint()

Test a character to see if it's any printable character, including a space

Synopsis:

```
#include <ctype.h>

int isprint( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isprint()* function tests for any printable character, including a space (' '). The [isgraph\(\)](#) (p. 1439) function is similar, except that the space character is excluded from the character set being tested.

Returns:

Nonzero if *c* is a printable character; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isprint( the_chars[i] ) ) {
            printf( "Char %c is a printable character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a printable character\n",
                    the_chars[i] );
        }
    }
}
```

```
        the_chars[i] );  
    }  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is a printable character  
Char  is not a printable character  
Char  is a printable character  
Char } is a printable character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

ispunct()

Test a character to see if it's any punctuation character

Synopsis:

```
#include <ctype.h>

int ispunct( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *ispunct()* function tests for any punctuation character such as a comma (,) or a period (.).

Returns:

Nonzero if *c* is punctuation; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', '!', '.', ',', ':', ';' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( ispunct( the_chars[i] ) ) {
            printf( "Char %c is a punctuation character\n",
                the_chars[i] );
        } else {
            printf( "Char %c is not a punctuation character\n",
                the_chars[i] );
        }
    }
}
```

```
    }  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is not a punctuation character  
Char ! is a punctuation character  
Char . is a punctuation character  
Char , is a punctuation character  
Char : is a punctuation character  
Char ; is a punctuation character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isspace()

Test a character to see if it's a whitespace character

Synopsis:

```
#include <ctype.h>

int isspace( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isspace()* function tests for the following whitespace characters:

' '

Space

'\f'

Form feed

'\n'

Newline or linefeed

'\r'

Carriage return

'\t'

Horizontal tab

'\v'

Vertical tab

The [isblank\(\)](#) (p. 1429) function tests for a narrower set of characters.

Returns:

Nonzero if *c* is a whitespace character; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d, '\n' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isspace( the_chars[i] ) ) {
            printf( "Char %c is a space character\n",
                the_chars[i] );
        } else {
            printf( "Char %c is not a space character\n",
                the_chars[i] );
        }
    }

    return EXIT_SUCCESS;
}
```

This program produces the output:

```
Char A is not a space character
Char  is a space character
Char  is a space character
Char } is not a space character
Char
is a space character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

isupper()

Test a character to see if it's an uppercase letter

Synopsis:

```
#include <ctype.h>

int isupper( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *isupper()* function tests for any uppercase letter A through Z.

Returns:

Nonzero if *c* is an uppercase letter; otherwise, zero.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

char the_chars[] = { 'A', 'a', 'z', 'Z' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;
    for( i = 0; i < SIZE; i++ ) {
        if( isupper( the_chars[i] ) ) {
            printf( "Char %c is an uppercase character\n",
                    the_chars[i] );
        } else {
            printf( "Char %c is not an uppercase character\n",
                    the_chars[i] );
        }
    }
}
```

```
    return EXIT_SUCCESS;
}
```

produces the output:

```
Char A is an uppercase character
Char a is not an uppercase character
Char z is not an uppercase character
Char Z is an uppercase character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

iswalnum()

Test a wide character to see if it's alphanumeric

Synopsis:

```
#include <wctype.h>

int iswalnum( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswalnum()* function tests if the argument *wc* is an alphanumeric wide character of the class `alpha` or `digit`. In the C locale, they're a to z, A to Z, 0 to 9.

Returns:

A nonzero value if the character is a member of the class `alpha` or `digit`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswalpha()

Test a wide character to see if it's alphabetic

Synopsis:

```
#include <wctype.h>

int iswalpha( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswalpha()* function tests if the argument *wc* is an alphabetic wide character of the class `a_lpha`. In the C locale, they are: a to z, A to Z.

Returns:

A nonzero value if the character is a member of the class `a_lpha`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswblank()

Test a wide character to see if it's a blank character

Synopsis:

```
#include <wctype.h>

int iswblank( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswblank()` function tests if the argument `wc` is a wide character of the class `blank`. In the C locale, this class includes the space character and `\t` (horizontal tab).

Returns:

A nonzero value if the character is a member of the class `blank`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswcntrl()

Test a wide character to see if it's a control character

Synopsis:

```
#include <wctype.h>

int iswcntrl( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswcntrl()* function tests if the argument *wc* is a control wide character of the class `cntrl`. In the C locale, this class consists of the ASCII characters from 0 through 31.

Returns:

A nonzero value if the character is a member of the class `cntrl`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswctype()

Test a wide character to see if it's a given character class

Synopsis:

```
#include <wctype.h>

int iswctype( wint_t wc,
              wctype_t charclass );
```

Arguments:

wc

The wide character you want to test.

charclass

The character class you want to test for. Get this class by calling [wctype\(\)](#) (p. 3722).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *iswctype()* function tests if the argument *wc* is a member of one or several character classes. For example:

This function:	Is equivalent to:
<code>iswalnum(wc)</code>	<code>iswctype(wc, wctype("alnum"))</code>
<code>iswalpha(wc)</code>	<code>iswctype(wc, wctype("alpha"))</code>
<code>ispunct(wc)</code>	<code>iswctype(wc, wctype("punct"))</code>



The results are unreliable if you didn't use *wctype()* to obtain *charclass*, or if a call to [setlocale\(\)](#) (p. 2880) affects `LC_CTYPE`.

Returns:

A nonzero value if the character is a member of the specified character class (or classes), or zero if the character isn't a member or *charclass* is 0.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswdigit()

Test a wide character to see if it's a decimal digit

Synopsis:

```
#include <wctype.h>

int iswdigit( wint_t wc );
```

Arguments:

wc

The wide character that you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be `WEOF`; otherwise, the function's behavior is undefined.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswdigit()` function tests if the argument `wc` is a decimal digit wide character of the class `digit`. In the C locale, this class consists of the characters 0 through 9.

Returns:

A nonzero value if the character is a member of the class `digit`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswgraph()

Test a wide character to see if it's any printable character except space

Synopsis:

```
#include <wctype.h>

int iswgraph( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswgraph()* function tests if the argument *wc* is a graphical wide character of the class `graph`. In the C locale, this class consists of all the printable characters, except the space character.

Returns:

A nonzero value if the character is a member of the class `graph`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswlower()

Test a wide character to see if it's a lowercase letter

Synopsis:

```
#include <wctype.h>

int iswlower( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswlower()* function tests if the argument *wc* is a lowercase wide character of the class `lower`. In the C locale, this class consists of the characters from `a` through `z`.

Returns:

A nonzero value if the character is a member of the class `lower`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswprint()

Test a wide character to see if it's any printable character, including space

Synopsis:

```
#include <wctype.h>

int iswprint( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswprint()* function tests if the argument *wc* is a printable wide character of the class `print`. In the C locale, this class consists of all the printable characters, including the space character.

Returns:

A nonzero value if the character is a member of the class `print`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswpunct()

Test a wide character to see if it's any punctuation character

Synopsis:

```
#include <wctype.h>

int iswpunct( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswpunct()` function tests if the argument `wc` is a punctuation wide character of the class `punct`. In the C locale, this class includes the comma (,) and the period (.), among others.

Returns:

A nonzero value if the character is a member of the class `punct`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswspace()

Test a wide character to see if it's a whitespace character

Synopsis:

```
#include <wctype.h>

int iswspace( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswspace()* function tests if the argument *wc* is a whitespace wide character of the class `space`. In the C locale, this class includes the space character, `\f` (form feed), `\n` (newline or linefeed), `\r` (carriage return), `\t` (horizontal tab), and `\v` (vertical tab).

Returns:

A nonzero value if the character is a member of the class `space`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswupper()

Test a character to see if it's an uppercase letter

Synopsis:

```
#include <wctype.h>

int iswupper( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *iswupper()* function tests if the argument *wc* is an uppercase wide character of the class `upper`. In the C locale, this class includes the characters from A through Z.

Returns:

A nonzero value if the character is a member of the class `upper`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

iswxdigit()

Test a wide character to see if it's a hexadecimal digit

Synopsis:

```
#include <wctype.h>

int iswxdigit( wint_t wc );
```

Arguments:

wc

The wide character you want to test. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswxdigit()` function tests if the argument `wc` is a hexadecimal wide character of the class `xdigit`. In the C locale, this class includes the characters 0 to 9, and A to F.

Returns:

A nonzero value if the character is a member of the class `xdigit`, or 0 otherwise.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The result is valid only for `wchar_t` arguments and `WEOF`.

isxdigit()

Test a character to see if it's a hexadecimal digit

Synopsis:

```
#include <ctype.h>

int isxdigit( int c );
```

Arguments:

c

The character you want to test. This must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *isxdigit()* function tests for any hexadecimal-digit character. These characters are the digits 0 through 9 and the letters a through f (or A through F).

Returns:

Nonzero if *c* is a hexadecimal digit; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', '5', '$' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isxdigit( the_chars[i] ) ) {
            printf( "Char %c is a hexadecimal digit",
                    the_chars[i] );
        } else {
            printf( "Char %c is not a hexadecimal digit",
                    the_chars[i] );
        }
    }
}
```

```
    }  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Char A is a hexadecimal digit character  
Char 5 is a hexadecimal digit character  
Char $ is not a hexadecimal digit character
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

itoa()

Convert an integer into a string, using a given base

Synopsis:

```
#include <stdlib.h>

char* itoa( int value,
            char* buffer,
            int radix );
```

Arguments:

value

The value to convert into a string.

buffer

A buffer in which the function stores the string. The size of the buffer must be at least:

$8 \times \text{sizeof}(\text{int}) + 1$

bytes when converting values in base 2 (binary).

radix

The base to use when converting the number.

If the value of *radix* is 10, and *value* is negative, then a minus sign is prepended to the result.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *itoa()* function converts the integer *value* into the equivalent string in base *radix* notation, storing the result in the specified *buffer*. The function terminates the string with a NUL character.

Returns:

A pointer to the resulting string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char buffer[20];
    int base;

    for( base = 2; base <= 16; base += 2 ) {
        printf( "%2d %s\n", base,
            itoa( 12765, buffer, base ) );
    }

    return EXIT_SUCCESS;
}
```

produces the output:

```
 2 11000111011101
 4 3013131
 6 135033
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Chapter 12

J

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “J”.

j0(), j0f()

Compute a Bessel function of the first kind

Synopsis:

```
#include <math.h>

double j0( double x );
float j0f( float x );
```

Arguments:

x

The number that you want to compute the Bessel function for.

Library:

libbessel

Use the -l bessel option to qcc to link against this library.

Description:

Compute the Bessel function of the first kind for *x*.

Returns:

The result of the Bessel function of *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;

    x = j0( 2.4 );
    y = y1( 1.58 );
    z = jn( 3, 2.4 );

    printf( "j0(2.4) = %f, y1(1.58) = %f\n", x, y );
```

```
printf( "jn(3,2.4) = %f\n", z );  
return EXIT_SUCCESS;  
}
```

Classification:

j0() is [POSIX 1003.1 XSI](#) (p. 46); jOf() is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

j1(), j1f()

Compute a Bessel function of the first kind

Synopsis:

```
#include <math.h>

double j1( double x );
float j1f( float x );
```

Arguments:

x

The number that you want to compute the Bessel function for.

Library:

libbessel

Use the -l bessel option to gcc to link against this library.

Description:

Compute the Bessel function of the first kind for *x*.

Returns:

The result of the Bessel function of *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

j1() is [POSIX 1003.1 XSI](#) (p. 46); *j1f()* is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

jn(), jnf()

Compute a Bessel function of the first kind

Synopsis:

```
#include <math.h>

double jn( int n, double x );
float jnf( int n, float x );
```

Arguments:

n, x

The numbers that you want to compute the Bessel function for.

Library:

libbessel

Use the -l bessel option to qcc to link against this library.

Description:

Compute the Bessel function of the first kind for *n* and *x*.

Returns:

The result of the Bessel function of *n* and *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;

    x = j0( 2.4 );
    y = y1( 1.58 );
    z = jn( 3, 2.4 );

    printf( "j0(2.4) = %f, y1(1.58) = %f\n", x, y );
```

```
    printf( "jn(3,2.4) = %f\n", z );  
    return EXIT_SUCCESS;  
}
```

Classification:

jn() is [POSIX 1003.1 XSI](#) (p. 46); *jnf()* is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

jrand48()

Generate a pseudo-random signed long integer in a thread-safe manner

Synopsis:

```
#include <stdlib.h>

long jrand48( unsigned short xsubi[3] );
```

Arguments:

xsubi

An array that comprises the 48 bits of the initial value that you want to use.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *jrand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a signed long integer uniformly distributed over the interval $[-2^{31}, 2^{31})$. It's a thread-safe version of [mrand48\(\)](#) (p. 1730).

The *xsubi* array should contain the desired initial value; this makes *jrand48()* thread-safe, and lets you start a sequence of random numbers at any known value.

Returns:

A pseudo-random long integer.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 13

K

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “K”.

kill()

Send a signal to a process or a group of processes

Synopsis:

```
#include <sys/types.h>
#include <signal.h>

int kill( pid_t pid,
          int sig );
```

Arguments:

pid

The ID of the process or process group that you want to send a signal to:

If <i>pid</i> is:	Then <i>sig</i> is sent to:
> 0	The single process with that process ID
0	All processes that are in the same process group as the sending process
< 0	Every process that's a member of the process group <i>-pid</i>

sig

Zero, or the signal that you want to send. For a complete list of signals, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *kill()* function sends the signal *sig* to a process or group of processes specified by *pid*. If *sig* is zero, no signal is sent, but the *pid* is still checked for validity.

For a process to have permission to send a signal to a process, either:

- the real or effective user ID of the sending process must match the real or effective user ID of the receiving process

Or:

- the calling process must have the `PROCmgr_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* isn't blocked, either *sig* or at least one pending unblocked signal is delivered before the *kill* function returns.

This call doesn't block. However, in the network case, lower priority threads may run.

Returns:

Zero, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EAGAIN

Insufficient system resources are available to deliver the signal.

EINVAL

The *sig* is invalid.

EPERM

The process doesn't have permission to send this signal to any receiving process; [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The given *pid* doesn't exist.

Examples:

See [sigprocmask\(\)](#) (p. 3004).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

killpg()

Send a signal to a process group

Synopsis:

```
#include <sys/types.h>
#include <signal.h>

int killpg( pid_t pgrp,
            int sig );
```

Arguments:

pid

The ID of the process group that you want to send a signal to.

sig

Zero, or the signal that you want to send. For a complete list of signals, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *killpg()* function sends the signal *sig* to the process group specified by *pgrp*. If *sig* is zero, no signal is sent, but *pgrp* is still checked for validity.

If *pgrp* is greater than 1, *killpg(pgrp, sig)* is equivalent to *kill(-pgrp, sig)*.

Returns:

Zero, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EAGAIN

Insufficient system resources are available to deliver the signal.

EINVAL

The signal *sig* is invalid or not supported.

EPERM

The process doesn't have permission to send this signal to any receiving process.

ESRCH

No process group can be found corresponding to the specified *pgrp* or *pgrp* is less than or equal to 1.

Examples:

See [sigprocmask\(\)](#) (p. 3004).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Chapter 14

L

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>LOG_MASK()</i>	setlogmask() (p. 2882)
<i>LOG_UPTO()</i>	setlogmask() (p. 2882)

This chapter describes the API components whose names begin with “L”.

labs()

Calculate the absolute value of a long integer

Synopsis:

```
#include <stdlib.h>

long labs( long j );
```

Arguments:

j

The number you want the absolute value of.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *labs()* function returns the absolute value of its long-integer argument *j*.

Returns:

The absolute value of *j*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    long x, y;

    x = -50000;
    y = labs( x );
    printf( "labs( %d ) = %d\n", x, y );
    return EXIT_SUCCESS;
}
```

produces the output:

```
labs( -50000 ) = 50000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lchown()

Change the user ID and group ID of a file or symbolic link

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

int lchown( const char * path,
            uid_t owner,
            gid_t group );
```

Arguments:

path

The name of the file whose ownership you want to change.

owner

The user ID of the new owner.

group

The group ID of the new owner.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *lchown()* function changes the user ID and group ID of the file specified by *path* to be the numeric values contained in *owner* and *group*, respectively. It's similar to [chown\(\)](#) (p. 292), except in the case where the named file is a symbolic link. In this case, *lchown()* changes the ownership of the symbolic link file itself, while *chown()* changes the ownership of the file or directory to which the symbolic link refers.

Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges (for example, the superuser) may change the ownership of a file.

In QNX Neutrino, the `_POSIX_CHOWN_RESTRICTED` flag is enforced. This means that only the superuser may change the ownership of a file. The group of a file may be changed by the superuser, or also by a process with the effective user ID equal to

the user ID of the file, if (and only if) *owner* is equal to the user ID of the file and *group* is equal to the effective group ID of the calling process.

If the *path* argument refers to a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode are cleared, if the function is successful.

If *lchown()* succeeds, the *st_ctime* field of the file is marked for update.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EACCES

Search permission is denied on a component of the path prefix.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of the *path* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT

A component of the path prefix doesn't exist, or the *path* arguments points to an empty string.

ENOSYS

The *lchown()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix isn't a directory.

EPERM

The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
/*
 * Change the ownership of a list of files
 * to the current user/group
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( lchown( argv[i], getuid(), getgid() ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }
    return( ecode );
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lcong48()

Initialize a sequence of pseudo-random numbers

Synopsis:

```
#include <stdlib.h>

void lcong48( unsigned short int param[7] );
```

Arguments:

param

An array of 7 short integers that are used to initialize the sequence;

- The first three entries are used to initialize the seed.
- The next three are used to initialize the multiplicand.
- The last entry is used to initialize the addend. You can't use values greater than 0xFFFF as the addend.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *lcong48()* function gives you full control over the multiplicand and addend used in *drand48()*, *erand48()*, *lrand48()*, *rand48()*, *mrnd48()*, and *jrand48()*, and the seed used in *drand48()*, *lrand48()*, and *mrnd48()*.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

ldexp(), ldexpf(), ldexpl()

Multiply a floating-point number by an integral power of 2

Synopsis:

```
#include <math.h>

double ldexp( double x,
              int exp );

float ldexp( float x,
            int exp );

long double ldexpl( long double x,
                   int exp );
```

Arguments:

x

A floating-point number.

exp

The exponent of 2 to multiply *x* by.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions multiply the floating-point number *x* by 2^{exp} .

A range error may occur.

Returns:

$x \times 2^{exp}$

For a correct value that would cause an underflow, these functions return 0.0.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    double value;

    value = ldexp( 4.7072345, 5 );
    printf( "%f\n", value );

    return EXIT_SUCCESS;
}
```

produces the output:

```
150.631504
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ldiv(), lldiv()

Perform division on long or long long integers

Synopsis:

```
#include <stdlib.h>

ldiv_t ldiv( long int numer,
             long int denom );

lldiv_t lldiv( long long numer,
               long long denom );
```

Arguments:

numer

The numerator.

denom

The denominator.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ldiv()* and *lldiv()* functions calculate the quotient and remainder of:

numer / denom

Returns:

A structure of type `ldiv_t` that contains the following members:

```
typedef struct {
    long int quot;    /* quotient */
    long int rem;     /* remainder */
} ldiv_t;
```

or a structure of type `lldiv_t` that contains the following members:

```
typedef struct {
    long long quot;    /* quotient */
    long long rem;     /* remainder */
} lldiv_t;
```


Examples:

```

#include <stdio.h>
#include <stdlib.h>

void print_time( long ticks )
{
    ldiv_t sec_ticks;
    ldiv_t min_sec;

    sec_ticks = ldiv( ticks, 100 );
    min_sec = ldiv( sec_ticks.quot, 60 );

    printf( "It took %d minutes and %d seconds.\n",
           min_sec.quot, min_sec.rem );
}

int main( void )
{
    print_time( 86712 );

    return EXIT_SUCCESS;
}

```

produces the output:

```
It took 14 minutes and 27 seconds.
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lfind()

Find an entry in a linear search table

Synopsis:

```
#include <search.h>

void * lfind( const void * key,
              const void * base,
              unsigned * num,
              unsigned width,
              int ( * compare)(
                const void * element1,
                const void * element2 ) );
```

Arguments:

key

The object to search for.

base

A pointer to the first element in the table.

num

A pointer to an integer containing the current number of elements in the table.

width

The size of an element, in bytes.

compare

A pointer to a user-supplied function that *lfind()* calls to compare an array element with the *key*. The arguments to the comparison function are:

- *element1* — the same pointer as *key*
- *element2* — a pointer to one of the array elements.

The comparison function must return 0 if *element1* equals *element2*, or a nonzero value if the elements aren't equal.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `lfind()` function returns a pointer into a table indicating where an entry may be found.



The `lfind()` function is the same as [lsearch\(\)](#) (p. 1552), except that if the entry isn't found, it isn't added to the table, and a `NULL` pointer is returned.

Returns:

A pointer to the matching element, or `NULL` if there's no match or an error occurred.

Examples:

This example program lets you know if the first command-line argument is a C keyword (assuming you fill in the `keywords` array with a complete list of C keywords):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <search.h>

static const char *keywords[] = {
    "auto",
    "break",
    "case",
    "char",
    /* . */
    /* . */
    /* . */
    "while"
};

int compare( const void *, const void * );

int main( int argc, const char *argv[] )
{
    unsigned num = 5;
    char *ptr;

    if( argc <= 1 ) return EXIT_FAILURE;

    ptr = lfind( &argv[1], keywords, &num, sizeof(char **), compare );
    if( ptr == NULL ) {
        printf( "%s' is not a C keyword\n", argv[1] );
        return EXIT_FAILURE;
    } else {
        printf( "%s' is a C keyword\n", argv[1] );
        return EXIT_SUCCESS;
    }

    /* You'll never get here. */
    return EXIT_SUCCESS;
}

int compare( const void *op1, const void *op2 )
{
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;

    return( strcmp( *p1, *p2 ) );
}
```

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lgamma(), lgamma_r(), lgammaf(), lgammaf_r()

Log gamma function

Synopsis:

```
#include <math.h>

double lgamma( double x );
double lgamma_r( double x,
                  int* signgamp);
float lgammaf( float x );
float lgammaf_r( float x,
                  int* signgamp);
```

Arguments:

x

An arbitrary number.

signgam

(*lgamma_r()*, *lgammaf_r()* only) A pointer to a location where the function can store the sign of *(x)*.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *lgamma()* and *lgamma_r()* functions return the natural log (ln) of the Γ function and are equivalent to [gamma\(\)](#) (p. 816). These functions return $\ln |\Gamma(x)|$, where $\Gamma(x)$ is defined as follows:

For $x > 0$:

$$\int_0^{\infty} e^{-t} t^{x-1} dt$$

For $x < 1$:

$$n / ((1-x) * \sin(n x))$$

The results converge when x is between 0 and 1. The `lgamma` function has the property:

$$\Gamma(N) = (N-1) \times \Gamma(N-1)$$

The `lgamma*` functions compute the log because the `gamma` function grows very quickly.

The `lgamma()` and `lgammaf()` functions use the external integer `signgam` to return the sign of $\Gamma(x)$, while `lgamma_r()` and `lgammaf_r()` use the user-allocated space addressed by `signgamp`.

The `signgam` variable isn't set until `lgamma()` or `lgammaf()` returns. For example, don't use the expression:



```
g = signgam * exp( lgamma( x ));
```

to compute $g = \Gamma(x)'$. Instead, compute `lgamma()` first:

```
lg = lgamma(x);
g = signgam * exp( lg );
```

Note that $\Gamma(x)$ must overflow when x is large enough, underflow when $-x$ is large enough, and generate a division by 0 exception at the singularities x a nonpositive integer.

Returns:

$\ln |\Gamma(x)|$



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` (p. 539) to 0, call the function, and then check `errno` again. These functions don't change `errno` if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;

    errno = EOK;
    a = 0.5;
    b = lgamma(a);
    printf("lgamma(%f) = %f %d \n", a, b, errno);

    return(0);
}
```

produces the output:

```
lgamma(0.500000) = 0.572365 0
```

Classification:

lgamma() and *lgammaf()* are [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *lgamma_r()* and *lgammaf_r()* are [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

link()

Create a link to an existing file

Synopsis:

```
#include <unistd.h>

int link( const char* existing,
          const char* new );
```

Arguments:

existing

The path of an existing file.

new

The path for the new link.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *link()* function creates a new directory entry named by *new* to refer to (that is, to be a link to) an existing file named by *existing*. The function atomically creates a new link for the existing file, and increments the link count of the file by one.



This implementation doesn't support using *link()* on directories or the linking of files across filesystems (different logical disks).

If the function fails, no link is created, and the link count of the file remains unchanged.

If *link()* succeeds, the *st_ctime* field of the file and the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EACCES

A component of either path prefix denies search permission, or the link named by *new* is in a directory with a mode that denies write permission.

EEXIST

The link named by *new* already exists.

ELOOP

Too many levels of symbolic links or prefixes.

EMLINK

The number of links to the file named by *existing* would exceed LINK_MAX.

ENAMETOOLONG

The length of the *existing* or *new* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT

This error code can mean the following:

- A component of either path prefix doesn't exist.
- The file named by *existing* doesn't exist.
- Either *existing* or *new* points to an empty string.

ENOSPC

The directory that would contain the link can't be extended.

ENOSYS

The *link()* function isn't implemented for the filesystem specified in *existing* or *new*.

ENOTDIR

A component of either path prefix isn't a directory.

EPERM

The file named by *existing* is a directory.

EROFS

The requested link requires writing in a directory on a read-only file system.

EXDEV

The link named by *new* and the file named by *existing* are on different logical disks.

Examples:

```
/*
 * The following program performs a rename
 * operation of argv[1] to argv[2].
 * Please note that this example, unlike the
 * library function rename(), ONLY works if
 * argv[2] doesn't already exist.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main( int argc, char** argv )
{
    /* Create a link of argv[1] to argv[2].
    */
    if( link( argv[1], argv[2] ) == -1 ) {
        perror( "link" );
        return( EXIT_FAILURE );
    }
    if( unlink( argv[1] ) == -1 ) {
        perror( argv[1] );
        return( EXIT_FAILURE );
    }
    return( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lio_listio()*, *lio_listio64()

Initiate a list of I/O requests

Synopsis:

```
#include <aio.h>

int lio_listio( int mode,
               struct aiocb* const list[],
               int nent,
               struct sigevent* sig );

int lio_listio64( int mode,
                 struct aiocb64* const list[],
                 int nent,
                 struct sigevent* sig );
```

Arguments:

mode

The mode of operation; one of:

- **LIO_WAIT** — behave synchronously, waiting until all I/O is completed, and ignore the *sig* argument.
- **LIO_NOWAIT** — behave asynchronously, returning immediately; the signal specified by the *sig* argument is delivered to the calling process when all the I/O operations from this function are completed.

list

An array of pointers to [aiocb](#) or [aiocb64](#) (p. 141) structures that specify the I/O operations that you want to initiate. The array may contain NULL pointers, which the function ignores.

nent

The number of entries in the *list* array. This must not exceed the system-wide limit, `_POSIX_AIO_MAX`.

sig

NULL, or a pointer to a [sigevent](#) (p. 2961) structure that specifies the signal that you want to deliver to the calling process when all of the I/O operations are completed. The function ignores this argument if *mode* is **LIO_WAIT**.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `lio_listio()` and `lio_listio64()` functions let the calling process, lightweight process (LWP), or thread initiate a list of I/O requests within a single function call. The `lio_listio64()` function is a large-file version of `lio_listio()`.

The `aio_lio_opcode` field of each `aiocb` structure in `list` specifies the operation to be performed (see `<aio.h>`):

- `LIO_READ` requests [aio_read\(\)](#) (p. 131).
- `LIO_WRITE` requests [aio_write\(\)](#) (p. 138).
- `LIO_NOP` causes the list entry to be ignored.

If `mode` is `LIO_NOWAIT`, `lio_listio()` and `lio_listio64()` use the [sigevent](#) (p. 2961) structure pointed to by `sig` to define both the signal to be generated and how the calling process is notified when the I/O operations are complete:

- If `sig` is `NULL`, or the `sigev_signo` member of the `sigevent` structure is zero, then no signal delivery occurs. Otherwise, the signal number indicated by `sigev_signo` is delivered when all the requests in the list have been completed.
- If `sig->sigev_notify` is `SIGEV_NONE`, no signal is posted upon I/O completion, but the error status and the return status for the operation are set appropriately.
- If `sig->sigev_notify` is `SIGEV_SIGNAL`, the signal specified in `sig->sigev_signo` is sent to the process. If the `SA_SIGINFO` flag is set for that signal number, the signal is queued to the process, and the value specified in `sig->sigev_value` is the `si_value` component of the generated signal.

For regular files, no data transfer occurs past the offset maximum established in the open file description associated with `aiocbp->aio_fildes`.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with `aio_fildes`. (see the definitions of `O_DSYNC` and `O_SYNC` in the description of [fcntl\(\)](#) (p. 623).)

Returns:

If the `mode` argument is `LIO_NOWAIT`, and the I/O operations are successfully queued, these functions return 0; otherwise they return -1 and set [errno](#) (p. 539).

If the `mode` argument is `LIO_WAIT`, and all the indicated I/O has been completed successfully, these functions return 0; otherwise, they return -1 and set `errno`.

In either case, the return value indicates only the success or failure of the *lio_listio()* call itself, not the status of the individual I/O requests. In some cases, one or more of the I/O requests contained in the list may fail. Failure of an individual request doesn't prevent completion of any other individual request. To determine the outcome of each I/O request, examine the error status associated with each *aio_cb* control block. Each error status so returned is identical to that returned as a result of calling *aio_read()* or *aio_write()*.

Errors:

EAGAIN

The resources necessary to queue all the I/O requests weren't available. The error status for each request is recorded in the *aio_error* member of the corresponding *aio_cb* structure, and can be retrieved using *aio_error()* (p. 127).

The number of entries, *nent*, exceeds the system-wide limit, *_POSIX_AIO_MAX*.

EINVAL

The *mode* argument is invalid.

The value of *nent* is greater than *_POSIX_AIO_LISTIO_MAX*.

EINTR

A signal was delivered while waiting for all I/O requests to be completed during an *LIO_WAIT* operation. However, the outstanding I/O requests aren't canceled. Use *aio_fsync()* (p. 129) to determine if any request was initiated; *aio_return()* (p. 134) to determine if any request has been completed; or *aio_error()* to determine if any request was canceled.

EIO

One or more of the individual I/O operations failed. Use *aio_error()* with each *aio_cb* structure to determine which request(s) failed.

ENOSYS

The *lio_listio()* function isn't supported by this implementation.

If either *lio_listio()* succeeds in queuing all of its requests, or *errno* is set to *EAGAIN*, *EINTR*, or *EIO*, then some of the I/O specified from the list may have been initiated. In this event, each *aio_cb* structure contains errors specific to the *read()* (p. 2502) or *write()* (p. 3737) function being performed:

EAGAIN

The requested I/O operation wasn't queued due to resource limitations.

ECANCELED

The requested I/O was canceled before the I/O was completed due to an explicit [aio_cancel\(\)](#) (p. 124) request.

EINPROGRESS

The requested I/O is in progress.

The following additional error codes may be set for each `aioctx` control block:

EOVERFLOW

The `aioctx->aio_lio_opcode` is `LIO_READ`, the file is a regular file, `aioctx->aio_nbytes` is greater than 0, and the `aioctx->aio_offset` is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with `aioctx->aio_fildes`.

EFBIG

The `aioctx->aio_lio_opcode` is `LIO_WRITE`, the file is a regular file, `aioctx->aio_nbytes` is greater than 0, and the `aioctx->aio_offset` is greater than or equal to the offset maximum in the open file description associated with `aioctx->aio_fildes`.

Classification:

`lio_listio()` is [POSIX 1003.1 AIO](#) (p. 46); `lio_listio64()` is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

listen()

Listen for connections on a socket

Synopsis:

```
#include <sys/socket.h>

int listen( int s,
            int backlog );
```

Arguments:

s

The descriptor for the socket that you want to listen on. You can create a socket by calling [socket\(\)](#) (p. 3075).

backlog

The maximum length that the queue of pending connections may grow to.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *listen()* function listens for connections on a socket and puts the socket into the LISTEN state. For connections to be accepted, you must:

1. Create a socket by calling *socket()*.
2. Indicate a willingness to accept incoming connections and a queue limit for them by calling *listen()*.
3. Call [accept\(\)](#) (p. 58) to accept the connections.

If a connection request arrives with the queue full, the client may receive an error with an indication of `ECONNREFUSED`. But if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.



The *listen()* call applies only to `SOCK_STREAM` sockets.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).**Errors:****EBADF**Invalid descriptor *s*.**EOPNOTSUPP**The socket isn't of a type that supports the *listen()* operation.**Classification:**[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

llabs()

Calculate the absolute value of a long long integer

Synopsis:

```
#include <stdlib.h>

long long llabs( long long j );
```

Arguments:

j

The number you want the absolute value of.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *llabs()* function returns the absolute value of its long-long-integer argument *j*.

Returns:

The absolute value of *j*.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

localeconv()

Set numeric formatting according to the current locale

Synopsis:

```
#include <locale.h>

struct lconv * localeconv( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *localeconv()* function gets the values appropriate for formatting numeric quantities using the current locale. It returns a pointer to a `struct lconv` with the following members:

char * *decimal_point*

The decimal-point character used for nonmonetary quantities.

char * *thousands_sep*

The character used to separate groups of digits on the left of the decimal-point character formatted nonmonetary quantities.

char * *int_curr_symbol*

The international currency symbol for the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217: Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the NUL character) is the character used to separate the international currency symbol from the monetary quantity.

char * *currency_symbol*

The local currency symbol applicable to the current locale.

char * *mon_decimal_point*

The decimal-point character used to format monetary quantities.

char * *mon_thousands_sep*

The character used to separate groups of digits on the left of the decimal-point character in formatted monetary quantities.

char * *mon_grouping*

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char * *grouping*

A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char * *positive_sign*

The string used to indicate a nonnegative monetary quantity.

char * *negative_sign*

The string used to indicate a negative monetary quantity.

char *int_frac_digits*

The number of fractional digits (to the right of the decimal point) to display in an internationally formatted monetary quantity.

char *frac_digits*

The number of fractional digits (to the right of the decimal point) to display in a formatted monetary quantity.

char *p_cs_precedes*

Set to 1 or 0 if the *currency_symbol* precedes or follows the value for a nonnegative monetary quantity.

char *p_sep_by_space*

Set to 1 or 0 if the *currency_symbol* is or isn't separated by a space from the value for a nonnegative monetary quantity.

char *n_cs_precedes*

Set to 1 or 0 if the *currency_symbol* precedes or follows the value for a negative monetary quantity.

char *n_sep_by_space*

Set to 1 or 0 if the *currency_symbol* is or isn't separated by a space from the value for a negative monetary quantity.

char *p_sign_posn*

The position of the *positive_sign* for a nonnegative monetary quantity.

char *n_sign_posn*

The position of the *positive_sign* for a negative monetary quantity.

The *grouping* and *mon_grouping* members have the following values:

CHAR_MAX

Perform no further grouping.

0

Repeat the previous element used for the remainder of the digits.

other

The value is the number of digits that comprise the current group. Examine the next element to determine the size of the next group of digits (to the left of the current group).

The *p_sign_posn* and *n_sign_posn* members have the following values:

0

Parentheses surround the quantity and *currency_symbol*.

1

The sign string precedes the quantity and *currency_symbol*.

2

The sign string follows the quantity and *currency_symbol*.

3

The sign string immediately precedes the quantity and *currency_symbol*.

4

The sign string immediately follows the quantity and *currency_symbol*.

Returns:

A pointer to the struct `lconv`.

Examples:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>
```

```

int main( void )
{
    struct lconv *lc;

    lc = localeconv();
    printf( "decimal_point (%s)\n",
        lc->decimal_point );

    printf( "thousands_sep (%s)\n",
        lc->thousands_sep );

    printf( "int_curr_symbol (%s)\n",
        lc->int_curr_symbol );

    printf( "currency_symbol (%s)\n",
        lc->currency_symbol );

    printf( "mon_decimal_point (%s)\n",
        lc->mon_decimal_point );

    printf( "mon_thousands_sep (%s)\n",
        lc->mon_thousands_sep );

    printf( "mon_grouping (%s)\n",
        lc->mon_grouping );

    printf( "grouping (%s)\n",
        lc->grouping );

    printf( "positive_sign (%s)\n",
        lc->positive_sign );

    printf( "negative_sign (%s)\n",
        lc->negative_sign );

    printf( "int_frac_digits (%d)\n",
        lc->int_frac_digits );

    printf( "frac_digits (%d)\n",
        lc->frac_digits );

    printf( "p_cs_precedes (%d)\n",
        lc->p_cs_precedes );

    printf( "p_sep_by_space (%d)\n",
        lc->p_sep_by_space );

    printf( "n_cs_precedes (%d)\n",
        lc->n_cs_precedes );

    printf( "n_sep_by_space (%d)\n",
        lc->n_sep_by_space );

    printf( "p_sign_posn (%d)\n",
        lc->p_sign_posn );

    printf( "n_sign_posn (%d)\n",
        lc->n_sign_posn );

    return EXIT_SUCCESS;
}

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

localtime()

Convert calendar time to local time

Synopsis:

```
#include <time.h>

struct tm *localtime( const time_t *timer );
```

Arguments:

timer

A pointer to a `time_t` object that contains the calendar time that you want to convert.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `localtime()` function converts the calendar time pointed to by *timer* into local time, storing the information in a `struct tm` (p. 3476). Whenever you call `localtime()`, it calls `tzset()` (p. 3547).

You typically get a calendar time by calling `time()` (p. 3425). That time is Coordinated Universal Time (UTC, formerly known as Greenwich Mean Time or GMT).



The `localtime()` function places the converted time in a static `tm` structure that's reused each time you call `localtime()` or `gmtime()` (p. 1005). Calling `asctime()` (p. 155) or `ctime()` (p. 389) could also change the data in this static buffer.

In a multithreaded application, use `localtime_r()` (p. 1533).

You typically use the `date` command to set the computer's internal clock using Coordinated Universal Time (UTC). Use the **TZ** environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Returns:

A pointer to the static [tm](#) (p. 3476) structure containing the time information.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

localtime_r()

Convert calendar time to local time

Synopsis:

```
#include <time.h>

struct tm* localtime_r( const time_t* timer,
                        struct tm* result );
```

Arguments:

timer

A pointer to a `time_t` object that contains the calendar time that you want to convert.

result

A pointer to a [tm](#) (p. 3476) structure where the function can store the converted time.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `localtime_r()` function converts the calendar time pointed to by *timer* into local time, storing the information in the `struct tm` (p. 3476) that *result* points to. Whenever you call `localtime_r()`, it calls [tzset\(\)](#) (p. 3547).

You typically get a calendar time by calling [time\(\)](#) (p. 3425). That time is Coordinated Universal Time (UTC, formerly known as Greenwich Mean Time or GMT).

You typically use the `date` command to set the computer's internal clock using Coordinated Universal Time (UTC). Use the **TZ** environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Returns:

A pointer to *result*, the `struct tm` (p. 3476).

Classification:

POSIX 1003.1 TSF (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lockf(), lockf64()

Lock or unlock a section of a file

Synopsis:

```
#include <unistd.h>

int lockf( int filedes,
           int function,
           off_t size );

int lockf64( int filedes,
             int function,
             off64_t size );
```

Arguments:

filedes

The file descriptor for the file that you want to lock. Open the file with write-only permission ([O_WRONLY](#) (p. ?)) or with read/write permission ([O_RDWR](#) (p. ?)).

function

A control value that specifies the action to be taken. The permissible values (defined in `<unistd.h>`) are as follows:

F_LOCK

Lock a section for exclusive use if the section is available. A read-only lock is one of [O_RDONLY](#) (p. ?), [O_WRONLY](#) (p. ?), or [O_RDWR](#) (p. ?). An exclusive lock is one of [O_WRONLY](#) (p. ?), or [O_RDWR](#) (p. ?). (For descriptions of the locks, see [open\(\)](#) (p. 1905)).

F_TEST

Test a specified section for locks obtained by other processes.

F_TLOCK

Test and lock a section for exclusive use if the section is available.

F_ULOCK

Remove locks from a specified section of the file.

size

The number of contiguous bytes that you want to lock or unlock. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive *size* or backward for a negative *size* (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest possible file offset is locked (that is, from the current offset through to the present or any future end-of-file). An area need not be allocated to the file to be locked because locks may exist past the end-of-file.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use the `lockf()` function to lock a section of a file, using advisory-mode locks. If other threads call `lockf()` to try to lock the locked file section, those calls either return an error value or block until the section becomes unlocked. The `lockf64()` function is a large-file version of `lockf()`.

All the locks for a process are removed when the process terminates. Record locking with `lockf()` is supported for regular files and may be supported for other files.

The sections locked with `F_LOCK` (p. ?) or `F_TLOCK` (p. ?) may in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections occur, the sections are combined into a single locked section.

File locks are released on the first close by the locking process of any file descriptor for the file.

`F_ULOCK` (p. ?) requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections are unlocked starting at the current file offset through *size* bytes or to the end of file if *size* is 0. When all of a locked section isn't released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. Releasing the center portion of a locked section causes the remaining locked beginning and end portions to become two separate locked sections.

A potential for deadlock occurs if the threads of a process controlling a locked section are blocked by accessing another process's locked section. If the system detects that deadlock could occur, `lockf()` fails with `EDEADLK`.

The interaction between *fcntl()* (p. 623) and *lockf()* locks is unspecified. Blocking on a section is interrupted by any signal.

If *size* is the maximum value of type *off_t* (or *off64_t* for *lockf64()*) and the process has an existing lock of size 0 in this range (indicating a lock on the entire file), then an *F_ULOCK* (p. ?) request is treated the same as an *F_ULOCK* (p. ?) request of size 0. Otherwise an *F_ULOCK* (p. ?) request attempts to unlock only the requested section. Attempting to lock a section of a file that's associated with a buffered stream produces unspecified results.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set). Existing locks aren't changed.

Errors:

EACCES or EAGAIN

The *function* argument is *F_TLOCK* (p. ?) or *F_TEST* (p. ?) and the section is already locked by another process.

EAGAIN

The *function* argument is *F_LOCK* (p. ?) or *F_TLOCK* (p. ?) and the file is mapped with *mmap()* (p. 1656).

EBADF

The *fildest* argument isn't a valid open file descriptor; or *function* is *F_LOCK* (p. ?) or *F_TLOCK* (p. ?) and *fildest* isn't a valid file descriptor open for writing.

EDEADLK

The *function* argument is *F_LOCK* (p. ?) and a deadlock is detected.

EINTR

A signal was caught during execution of the function.

EINVAL

The *function* argument isn't one of *F_LOCK* (p. ?), *F_TLOCK* (p. ?), *F_TEST* (p. ?) or *F_ULOCK* (p. ?); or *size* plus the current file offset is less than 0.

ENOMEM

The system can't allocate sufficient memory to store lock resources.

ENOSYS

The filesystem doesn't support the operation.

EOPNOTSUPP or EINVAL

The implementation doesn't support the locking of files of the type indicated by *files*.

EOVERFLOW

The offset of the first, or if *size* isn't 0 then the last, byte in the requested section can't be represented correctly in an object of type `off_t`.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

log(), logf(), logl()

Compute the natural logarithm of a number

Synopsis:

```
#include <math.h>

double log( double x );
float logf( float x );
long double logl( long double x );
```

Arguments:

x

The number that you want to compute the natural log of.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *log()* and *logf()* functions compute the natural logarithm (base *e*) of *x*:

$\log_e x$

A domain error occurs if *x* is negative. A range error occurs if *x* is zero.

Returns:

The natural logarithm of *x*. For finite values of *x* less than zero, these functions return NaN. The return value when *x* is `-Inf` is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
int main( void )
{
    printf( "%f\n", log(.5) );

    return EXIT_SUCCESS;
}
```

produces the output:

-0.693147

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

log10(), log10f(), log10l()

Compute the logarithm (base 10) of a number

Synopsis:

```
#include <math.h>

double log10( double x );
float log10f( float x );
long double log10l( long double x );
```

Arguments:

x

The number that you want to compute the log of.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the base 10 logarithm of *x*:

$\log_{10} x$

A domain error occurs if *x* is negative. A range error occurs if *x* is zero.

Returns:

The base 10 logarithm of *x*. For finite values of *x* less than zero, these functions return NaN. The return value when *x* is -Inf is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
int main( void )
{
    printf( "%f\n", log10(.5) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
-0.301030
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

log1p(), log1pf(), log1pl()

 $\log(1+x)$ **Synopsis:**

```
#include <math.h>

double log1p ( double x );
float log1pf ( float x );
long double log1pl( long double x );
```

Arguments:*x*

The number that you want to add 1 to and compute the natural log of.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *log1p()*, *log1pf()*, and *log1pl()* functions compute the value of $\log(1+x)$, where $x > -1.0$.

Returns: $\log(1+x)$

For finite values of x less than -1, these functions return NaN. The return value when x is -Inf is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

logb(), logbf()

Compute the radix-independent exponent

Synopsis:

```
#include <math.h>

double logb ( double x );
float logbf ( float x );
```

Arguments:

x

The number that you want to compute the radix-independent exponent of.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *logb()* and *logbf()* functions compute the exponent part of *x*, which is the integral part of:

$\log_r |x|$

as a signed floating point value, for nonzero finite *x*, where *r* is the radix of the machine's floating point arithmetic.

Returns:

The binary exponent of *x*, a signed integer converted to double-precision floating-point.

If <i>x</i> is:	<i>logb()</i> returns:
0.0	-HUGE_VAL (<i>errno</i> is set to EDOM)
<0.0	-HUGE_VAL (<i>errno</i> may be set to ERANGE)
±infinity	+infinity



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set *errno* (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;

    a = 0.5;
    b = logb(a);
    printf("logb(%f) = %f (%f = 2^%f) \n", a, b, a, b);

    return(0);
}
```

produces the output:

```
logb(0.500000) = -1.000000 (0.500000 = 2^-1.000000)
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

login_tty()

Prepare for a login in a tty

Synopsis:

```
#include <unix.h>

int login_tty( int fd );
```

Arguments:

fd

A file descriptor that you want to use as the controlling terminal for the current process.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `login_tty()` function prepares for a login on the tty `fd` (which may be a real tty device, or the slave of a pseudo-tty as returned by [openpty\(\)](#) (p. 1921)) by creating a new session, making `fd` the controlling terminal for the current process, setting `fd` to be the standard input, output, and error streams of the current process, and closing `fd`.

This function fails if [ioctl\(\)](#) (p. 1198) fails to set `fd` to the controlling terminal of the current process.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Classification:*Unix* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

longjmp()

Restore the environment saved by setjmp()

Synopsis:

```
#include <setjmp.h>

void longjmp( jmp_buf env,
              int return_value );
```

Arguments:

env

The environment saved by the most recent call to [setjmp\(\)](#) (p. 2875).

return_value

The value that you want *setjmp()* to return.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *longjmp()* function restores the environment saved in *env* by the most recent call to the [setjmp\(\)](#) (p. 2875) function.



Using *longjmp()* to jump out of a signal handler can cause unpredictable behavior, unless the signal was generated by the [raise\(\)](#) (p. 2482) function.

Returns:

After the *longjmp()* function restores the environment, program execution continues as if the corresponding call to *setjmp()* had just returned the value specified by *return_value*. If the value of *return_value* is 0, the value returned is 1.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;
```

```
void rtn( void )
{
    printf( "about to longjmp\n" );
    longjmp( env, 14 );
}

int main( void )
{
    int ret_val = 293;

    if( 0 == ( ret_val = setjmp( env ) ) ) {
        printf( "after setjmp %d\n", ret_val );
        rtn();
        printf( "back from rtn %d\n", ret_val );
    } else {
        printf( "back from longjmp %d\n", ret_val );
    }

    return EXIT_SUCCESS;
}
```

produces the following output:

```
after setjmp 0
about to longjmp
back from longjmp 14
```

Classification:

[ANSI, POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

A strictly-conforming POSIX application can't assume that the *longjmp()* function is signal-safe on other platforms.



Don't use [longjmp\(\)](#) or [siglongjmp\(\)](#) (p. 2970) to restore an environment saved by a call to [setjmp\(\)](#) (p. 2875) or [sigsetjmp\(\)](#) (p. 3010) in another thread. If you're lucky, your application will crash; if not, it'll look as if it works for a while, until random scribbling on the stack causes it to crash.

lrand48()

Generate a pseudo-random nonnegative long integer

Synopsis:

```
#include <stdlib.h>

long lrand48( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *lrand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative Long integer uniformly distributed over the interval $[0, 2^{31}]$.

Call one of [lcong48\(\)](#) (p. 1505), [seed48\(\)](#) (p. 2787), or [srand48\(\)](#) (p. 3157) to initialize the random-number generator before calling [drand48\(\)](#) (p. 468), *lrand48()*, or [mrand48\(\)](#) (p. 1730),

The [nrand48\(\)](#) (p. 1894) function is a thread-safe version of *lrand48()*.

Returns:

A pseudo-random Long integer.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

lsearch()

Perform a linear search in an array

Synopsis:

```
#include <search.h>

void * lsearch( const void * key,
               const void * base,
               unsigned * num,
               unsigned width,
               int ( * compare)(
                   const void * element1,
                   const void * element2 ) );
```

Arguments:

key

The object to search for.

base

A pointer to the first element in the table.

num

A pointer to an integer containing the current number of elements in the table.

width

The size of an element, in bytes.

compare

A pointer to a user-supplied function that *lsearch()* calls to compare an array element with the *key*. The arguments to the comparison function are:

- *element1* — the same pointer as *key*
- *element2* — a pointer to one of the array elements.

The comparison function must return 0 if *element1* equals *element2*, or a nonzero value if the elements aren't equal.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `lsearch()` function searches a linear table and returns a pointer into the table indicating where the entry was found.



If *key* isn't found, it's added to the end of the array and *num* is incremented.

Returns:

A pointer to the element that was found or created, or `NULL` if an error occurred.

Examples:

This program builds an array of pointers to the *argv* arguments by searching for them in an array of `NULL` pointers. Because none of the items will be found, they'll all be added to the *array*.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <search.h>

int compare( const void *, const void * );

int main( int argc, const char **argv )
{
    int i;
    unsigned num = 0;
    char **array = (char **)calloc( argc, sizeof(char **) );
    for( i = 1; i < argc; ++i ) {
        lsearch( &argv[i], array, &num, sizeof(char **), compare );
    }
    for( i = 0; i < num; ++i ) {
        printf( "%s\n", array[i] );
    }
    return EXIT_SUCCESS;
}

int compare( const void *op1, const void *op2 )
{
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;
    return( strcmp( *p1, *p2 ) );
}
```

Using the program above, this input:

```
one two one three four
```

produces the output:

```
one
two
three
four
```

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

lseek(), lseek64()

Set the current file position at the operating-system level

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek( int filedes,
             off_t offset,
             int whence );

off64_t lseek64( int filedes,
                 off64_t offset,
                 int whence );
```

Arguments:

filedes

The file descriptor of the file whose position you want to set.

offset

The relative offset from the file position determined by the *whence* argument.

whence

The position in the file. The possible values (defined in <unistd.h>) are:

SEEK_CUR

The new file position is computed relative to the current file position. The value of *offset* may be positive, negative or zero.

SEEK_END

The new file position is computed relative to the end of the file.

SEEK_SET

The new file position is computed relative to the start of the file. The value of *offset* must not be negative.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `lseek()` and `lseek64()` functions set the current file position for the file descriptor specified by *filedes* at the operating system level. The `lseek64()` function is a 64-bit version of `lseek()`. File descriptors are returned by a successful execution of one of the [creat\(\)](#) (p. 381), [dup\(\)](#) (p. 483), [dup2\(\)](#) (p. 485), [fcntl\(\)](#) (p. 623), [open\(\)](#) (p. 1905) or [sopen\(\)](#) (p. 3083) functions.

An error occurs if the requested file position is before the start of the file.

If the requested file position is beyond the end of the file and data is written at this point, subsequent reads of data in the gap will return bytes whose value is equal to zero (`'\0'`) until data is actually written into the gap.

These functions don't extend the size of a file (see [chsize\(\)](#) (p. 298)).

Returns:

The current file position, with 0 indicating the start of the file, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**EBADF**

The *filedes* argument isn't a valid file descriptor.

EINVAL

The *whence* argument isn't a proper value, or the resulting file offset is invalid.

ENOSYS

The `lseek()` function isn't implemented for the filesystem specified by *filedes*.

EOVERFLOW

The resulting file offset is a value that can't be represented correctly in an object of type `off_t`.

ESPIPE

The *filedes* argument is associated with a pipe or FIFO.

Examples:

Using the `lseek()` function, you can get the current file position (in fact, [tell\(\)](#) (p. 3373) is implemented this way). You can then use this value with another call to `lseek()` to reset the file position:

```
off_t file_posn;
int filedes;

/* get current file position */
file_posn = lseek( filedes, 0L, SEEK_CUR );

...

/* return to previous file position */
file_posn = lseek( filedes, file_posn, SEEK_SET );
```

If all records in the file are the same size, the position of the *n*th record can be calculated and read like this:

```
#include <sys/types.h>
#include <unistd.h>

int read_record( int filedes, long rec_num,
                 int rec_size, char *buffer )
{
    if( lseek( filedes, rec_num * rec_size,
               SEEK_SET ) == -1L ) {
        return -1;
    }
    return( read( filedes, buffer, rec_size ) );
}
```

The `read_record()` function in this example assumes records are numbered starting with zero, and that `rec_size` contains the size of a record in the file, including any record-separator characters.

Classification:

`lseek()` is [POSIX 1003.1](#) (p. 46); `lseek64()` is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

lstat(), *lstat64()*

Get information about a file or directory

Synopsis:

```
#include <sys/stat.h>

int lstat( const char* path,
           struct stat* buf );

int lstat64( const char* path,
             struct stat64* buf );
```

Arguments:

path

The path of the file or directory that you want information about.

buf

A pointer to a buffer where the function can store the information.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *lstat()* and *lstat64()* functions obtain information about the file or directory referenced in *path*. This information is placed in the structure located at the address indicated by *buf*. The *lstat64()* function is a 64-bit version of *lstat()*.

The results of the *lstat()* function are the same as the results of [stat\(\)](#) (p. 3164) when used on a file that isn't a symbolic link. If the file is a symbolic link, *lstat()* returns information about the symbolic link, while *stat()* continues to resolve the pathname using the contents of the symbolic link, and returns information about the resulting file.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

See [stat\(\)](#) (p. 3164) for details.

Examples:

```
/*
 * Iterate through a list of files, and report
 * for each if it is a symbolic link
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    int ecode = 0;
    int n;
    struct stat sbuf;

    for( n = 1; n < argc; ++n ) {
        if( lstat( argv[n], &sbuf ) == -1 ) {
            perror( argv[n] );
            ecode++;
        } else if( S_ISLNK( sbuf.st_mode ) ) {
            printf( "%s is a symbolic link\n", argv[n] );
        } else {
            printf( "%s is not a symbolic link\n", argv[n] );
        }
    }
    return( ecode );
}
```

Classification:

lstat() is [POSIX 1003.1](#) (p. 46); *lstat64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

ltoa(), lltoa()

Convert a long integer into a string, using a given base

Synopsis:

```
#include <stdlib.h>

char* ltoa( long value,
            char* buffer,
            int radix );

char* lltoa( long long value,
             char* buffer,
             int radix );
```

Arguments:

value

The value to convert into a string.

buffer

A buffer in which the function stores the string. The size of the buffer must be at least 33 bytes when converting values in base 2 (binary).

radix

The base to use when converting the number. This value must be in the range:

2 *radix* 36

If the value of *radix* is 10, and *value* is negative, then a minus sign is prepended to the result.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *ltoa()* and *lltoa()* functions convert the given long integer *value* into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A NUL character is appended to the result.

Returns:

A pointer to the result.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

void print_value( long value )
{
    int base;
    char buffer[33];

    for( base = 2; base <= 16; base = base + 2 ) {
        printf( "%2d %s\n", base,
                ltoa( value, buffer, base ) );
    }
}

int main( void )
{
    print_value( 12765 );

    return EXIT_SUCCESS;
}
```

produces the output:

```
2 11000111011101
4 3013131
6 135033
8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

ltoa() is [QNX 4](#) (p. 46); *lltoa()* is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

ltrunc()

Truncate a file at a given position

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

off_t ltrunc( int fildes,
              off_t offset,
              int whence );
```

Arguments:

fildes

The file descriptor of the file that you want to truncate.

offset

The relative offset from the file position determined by the *whence* argument.

whence

The position in the file. The possible values (defined in <unistd.h>) are:

SEEK_CUR

The new file position is computed relative to the current file position. The value of *offset* may be positive, negative or zero.

SEEK_END

The new file position is computed relative to the end of the file.

SEEK_SET

The new file position is computed relative to the start of the file. The value of *offset* must not be negative.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *ltruncate()* function attempts to truncate the file at a specified position. The file, referenced by the open file descriptor *fd*, must have been opened **O_WRONLY** or **O_RDWR**. The truncation point is calculated using the value of *offset* as a relative offset from a file position determined by the value of the argument *whence*. The value of *offset* may be negative, although a negative truncation point (one before the beginning of the file) is an error.



The *ltruncate()* function ignores advisory locks that may have been set by *fcntl()* (p. 623).

The calculated truncation point, if within the existing bounds of the file, determines the new file size; all data after the truncation point no longer exists. If the truncation point is past the existing end of file, the file size isn't changed. An error occurs if you attempt to truncate before the beginning of the file (that is, a negative truncation point).



The current seek position isn't changed by this function under any circumstance, including the case where the current seek position is beyond the truncation point.

Returns:

Upon successful completion, this function returns the new file size. If a truncation point beyond the existing end of file was specified, the existing file size is returned, and the file size remains unchanged. Otherwise, *ltruncate()* returns a value of -1 and sets *errno* (p. 539) to indicate the error. The file size remains unchanged in the event of an error.

Errors:**EBADF**

The *fd* argument isn't a valid file descriptor, open for writing.

EINVAL

The *whence* argument isn't a proper value, or the resulting file size would be invalid.

ENOSYS

An attempt was made to truncate a file of a type that doesn't support truncation (for example, a file associated with the device manager).

ESPIPE

The *files* argument is associated with a pipe or FIFO.

Examples:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

char buffer[1000];

int main( void )
{
    int fd, stat;

    fd = open( "test", O_CREAT | O_RDWR, 0666 );
    if( fd == -1 ) {
        fprintf( stderr, "Open error\n" );
        exit( -1 );
    }

    /* Create a 1000-byte file */
    write( fd, buffer, 1000 );

    /* Seek back to offset 500 and truncate the file */
    if( ltrunc( fd, 500, SEEK_SET ) == -1 ) {
        fprintf( stderr, "ltrunc error\n" );
        exit( -1 );
    }
    close( fd );
    fd = open( "test", O_CREAT | O_RDWR, 0666 );
    printf( "File size = %ld\n",
        lseek( fd, 0, SEEK_END ) );
    close( fd );

    return 0;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The *ltrunc()* function *isn't portable*, and shouldn't be used in new code. Use [ftruncate\(\)](#) (p. 793) instead.

Chapter 15

M

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>major()</i>	stat() (p. 3164)
<i>makedev()</i>	stat() (p. 3164)
<i>minor()</i>	stat() (p. 3164)

This chapter describes the API components whose names begin with “M”.

main()

Program entry function

Synopsis:

```
int main( void );  
  
int main( int argc,  
          const char *argv[] );  
  
int main( int argc,  
          const char *argv[],  
          char *envp[] );
```

Arguments:

The arguments depend on which form of *main()* that you use.

argc

The number of entries in the *argv* array.

argv

An array of pointers to strings that contain the arguments to the program.

envp

An array of pointers to strings that define the environment for the program.

Description:

The *main()* function is supplied by the user and is where program execution begins. The command line to the program is broken into a sequence of tokens separated by blanks, and are passed to *main()* as an array of pointers to character strings in *argv*. The number of arguments found is passed in the parameter *argc*.

The *argv[0]* argument is a pointer to a character string containing the program name. The last element of the array pointed to by *argv* is NULL (*argv[argc]* is NULL). Arguments containing blanks can be passed to *main()* by enclosing them in quote characters (which are removed from that element in the *argv* vector). See your shell's documentation for details.

The *envp* argument points to an array of pointers to character strings that are the environment strings for the current process. This value is identical to the [environ](#) (p. 528) variable, which is defined in the `<stdlib.h>` header file.

Returns:

A value back to the calling program (usually the operating system).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    int i;
    for( i = 0; i < argc; ++i ) {
        printf( "argv[%d] = %s\n", i, argv[i] );
    }
    return EXIT_SUCCESS;
}
```

produces the output:

```
argv[0] = ./mypgm
argv[1] = hhhhh
argv[2] = another arg
```

when the program mypgm is run from the shell:

```
$ ./mypgm hhhhh "another arg"
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mallinfo()

Get memory allocation information

Synopsis:

```
#include <malloc.h>

struct mallinfo mallinfo ( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mallinfo()* function returns memory-allocation information in the form of a `struct mallinfo`:

```
struct mallinfo {
    int arena; /* size of the arena */
    int ordblks; /* number of big blocks in use */
    int smblks; /* number of small blocks in use */
    int hblks; /* number of header blocks in use */
    int hblkhd; /* space in header block headers */
    int usmblks; /* space in small blocks in use */
    int fsmblks; /* memory in free small blocks */
    int uordblks; /* space in big blocks in use */
    int fordblks; /* memory in free big blocks */
    int keepcost; /* penalty if M_KEEP is used
                  -- not used */
};
```

Returns:

A `struct mallinfo`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

malloc()

Allocate memory

Synopsis:

```
#include <stdlib.h>

void* malloc( size_t size );
```

Arguments:

size

The number of bytes to allocate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *malloc()* function allocates a buffer of *size* bytes. Use [free\(\)](#) (p. 717) or [realloc\(\)](#) (p. 2527) to free the block of memory.



Because the *malloc()* implementation uses signed, 32-bit integers to represent the size internally, you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, *malloc()* indicates an error of `ENOMEM`.

If *size* is zero, the default behavior is to return a non-NULL pointer that's valid only to a corresponding call to *free()* or *realloc()*. Don't assume that this pointer points to any valid memory. You can control this behavior via the **MALLOC_OPTIONS** environmental variable; if the value of **MALLOC_OPTIONS** contains a V, *malloc()* returns a NULL pointer. This environment variable also affects [calloc\(\)](#) (p. 255) and *realloc()*. This is known as the “System V” behavior.



If there isn't a free block of memory that's large enough to satisfy the request, the memory allocator uses [mmap\(\)](#) (p. 1656) to get memory from the system. The `_ambblksiz` (p. 151) global variable (defined in `<stdlib.h>`) specifies the number of bytes that the allocator gets from the system. You can also use the [mallopt\(\)](#) (p. 1575) function to control how the system allocates memory.

The heap is protected by such security measures as address space layout randomization (ASLR), which randomizes the stack start address and code locations in executables and libraries, and heap cookies.

Returns:

A pointer to the start of the allocated memory, or `NULL` if an error occurred (*errno* (p. 539) is set).

Errors:**ENOMEM**

Not enough memory.

EOK

No error.

Examples:

```
#include <stdlib.h>

int main( void )
{
    char* buffer;

    buffer = (char* )malloc( 80 );
    if( buffer != NULL ) {
        /* do something with the buffer */
        ...
        free( buffer );
    }

    return EXIT_SUCCESS;
}
```

Environment variables:

You can modify the allocator characteristics by setting the following environment variables, which control how memory is cached by the allocator, and when it's released back to the system:

MALLOC_ARENA_CACHE_MAXBLK

The maximum number of cached arena blocks (the cache is used to track freed blocks that are held and not released back to the OS). The default value is 12.

MALLOC_ARENA_CACHE_MAXSZ

The maximum size of the arena cache.

MALLOC_ARENA_SIZE

The size of the *malloc()* arena. It also controls the amount by which the arena grows.

By default, this is set to 32 KB, which results in a minimum allocation of 64 KB by the allocator. You can lower this as long as it's a multiple of the page size (4096). The allocator will still attempt to get memory from the system in multiples of this value, and since it creates buckets of varying sizes, it may choose to allocate core memory for a bucket of blocks of a new size, even though an earlier allocation of core memory for a block of a different size hasn't been fully exhausted.

Memory in the allocator is allocated with different policies for blocks smaller than 128 bytes and blocks larger than 128 bytes:

- Smaller blocks are allocated using a fixed chunk allocator, which gets system memory in bunches of pieces of the size specified by **MALLOC_ARENA_SIZE** and creates smaller buckets from them.
- Larger memory blocks are created by also getting core memory from the system in multiples of **MALLOC_ARENA_SIZE** and carving out the appropriate user pieces from this, putting the remaining space onto a free list.

So, for example, if you make a simple pair of allocations such as:

```
malloc(100);
malloc(200);
```

the system creates a 32 KB arena allocation for the bucket size of 128 bytes to service the 100-byte allocation, and a 32 KB arena allocation to service the 200-byte allocation.

Lowering **MALLOC_ARENA_SIZE** can reduce the amount of memory required.

MALLOC_FREE_LIFO

If set, *malloc()* changes the free queueing strategy from the default (FIFO) to LIFO.

MALLOC_MEMORY_HOLD

If this environment variable is nonzero, the heap will never shrink; freed blocks are never released back to the OS, but are maintained in the arena cache.

MALLOC_MMAP_NOZERO

If this environment variable is nonzero, then when *malloc()* needs to expand the heap, it specifies the MAP_NOINIT flag when it calls *mmap()* (p. 1656); if the physical memory being mapped was previously unmapped with

UNMAP_INIT_OPTIONAL, then the POSIX requirement that the memory be zeroed is relaxed. For more information, see “Initializing allocated memory” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

There also other ways to control the exact distribution of sizes used to create small bands of memory and the threshold at which the smaller buckets end and the list-based allocator for larger blocks begins. This is also done with environment variables:

MALLOC_BAND_CONFIG_STR

Use this environment variable to configure the bands of memory. The string format is:

N: s1, n1, p1: s2, n2, p2: s3, n3, p3: . . . : sN, nN, pN

where the components are:

s

The band size.

n

The number of items.

p

The preallocation value, which can be zero.

MALLOC_MEMORY_PREALLOCATE

Preallocate and populate the arena cache by setting this environment variable to a value that specifies the size of the total arena cache.

MALLOC_OPTIONS

Control the way *calloc()*, *malloc()*, and *realloc()* behave if you specify a size of 0 (or a value of 0 for the *n* argument to *calloc()*). The V (“System V”) and R (“use the *realloc()* behavior of QNX Neutrino 6.4.0 and earlier”) columns below indicate how the functions behave if the value of **MALLOC_OPTIONS** includes that letter:

Function	Default	V	R
<i>calloc</i> (<i>n</i> , 0)	Non-NULL	NULL	No effect
<i>malloc</i> (0)	Non-NULL	NULL	No effect
<i>realloc</i> (NULL, 0)	Non-NULL	NULL	No effect

Function	Default	V	R
<i>realloc(non-NULL, 0)</i>	Non-NULL	NULL	NULL

In all the above cases, if the function returns a non-NULL pointer, it's valid only for a corresponding call to *free()* or *realloc()*.

For more information, see “Dynamic memory management” in the Heap Analysis: Making Memory Errors a Thing of the Past chapter of the QNX Neutrino *Programmer's Guide*.

Debugging

The debug malloc library also uses these environment variables:

MALLOC_INITVERBOSE

Enable some initial verbose output regarding other variables that are enabled.

MALLOC_BTDEPTH

The depth of the backtrace for allocations (i.e. where the allocation occurred) on CPUs that support deeper backtrace levels. Currently the builtin-return-address feature of gcc is used to implement deeper backtraces for the debug malloc library. The default value is 0.

MALLOC_TRACEBT

Set the depth of the backtrace for errors and warnings on CPUs that support deeper backtrace levels. Currently the builtin-return-address feature of gcc is used to implement deeper backtraces for the debug malloc library. The default value is 0.

MALLOC_DUMP_LEAKS

Trigger leak detection on exit of the program. The output of the leak detection is sent to the file named by this variable.

MALLOC_TRACE

Enable tracing of all calls to *malloc()*, *free()*, *calloc()*, *realloc()*, etc. A trace of the various calls is store in the file named by this variable.

MALLOC_CHKACCESS_LEVEL

Specify the level of checking performed by the MALLOC_CHKACCESS option to *mallopt()*.

Reverting to the behavior of Neutrino 6.2

In order to revert to the allocator behavior from QNX Neutrino 6.2, do one of the following:

- Set **`MALLOC_ARENA_CACHE_MAXBLK`** to 0, which results in no caching of arenas.

Or:

- Set **`MALLOC_ARENA_CACHE_MAXSZ`** to 400000, which limits the total caching of the blocks to 400000 bytes.

You can also set both variables to fine-tune the allocator's behavior. The easiest way is to explicitly set the **`MALLOC_ARENA_CACHE_MAXSZ`** value to set a cap on the total amount of memory held in the allocator. The **`MALLOC_ARENA_CACHE_MAXBLK`** value is more difficult to configure, because arena blocks aren't necessarily the same size, and hence setting the number alone may not give the exact desired behavior.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Don't use [brk\(\)](#) (p. 213) and [sbrk\(\)](#) (p. 2702) with any other memory functions (such as [malloc\(\)](#), [mmap\(\)](#) (p. 1656), and [free\(\)](#) (p. 717)). The [brk\(\)](#) function assumes that the heap is contiguous; in QNX Neutrino, memory is returned to the system by the heap, causing the heap to become sparse. The QNX Neutrino [malloc\(\)](#) function is based on [mmap\(\)](#), and not on [brk\(\)](#).

In QNX 4, nothing is allocated when you [malloc\(\)](#) 0 bytes. Be careful if your code is ported between QNX 4 and later QNX operating systems.

mallopt()

Control the extra checking for memory allocation



The debug version of this function is declared in `<rcheck/malloc.h>`. The debug version includes additional commands and settings that aren't in `<malloc/malloc.h>`. To use *debug mallopt()*, you need to link with `librcheck.so` or use `LD_PRELOAD=librcheck.so` at runtime.

Synopsis:

```
#include <malloc/malloc.h>

int mallopt( int cmd,
             int value );
```

Arguments:

cmd

The command or setting you wish to make. The normal and debug version of *mallopt()* both support the following:

- MALLOC_ARENA_SIZE
- MALLOC_ARENA_CACHE_FREE_NOW
- MALLOC_ARENA_CACHE_MAXBLK
- MALLOC_ARENA_CACHE_MAXSZ
- MALLOC_FREE_CHECK
- MALLOC_MEMORY_HOLD
- MALLOC_STATS
- MALLOC_VERIFY_ON

The debug version also supports the following (which the normal version ignores):

- MALLOC_CKACCESS
- MALLOC_CKALLOC
- MALLOC_CKBOUNDS
- MALLOC_CKCHAIN
- MALLOC_DUMP_LEAKS
- MALLOC_EVENTBTDEPTH
- MALLOC_START_TRACING
- MALLOC_TRACEMIN

- `MALLOC_TRACEMAX`
- `MALLOC_TRACEBTDEPTH`
- `MALLOC_TRACING`
- `MALLOC_VERIFY`
- `MALLOC_FATAL`
- `MALLOC_WARN`

value

A value corresponding to the command.

For more information about the commands and their possible values, see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `malloc()` function controls memory allocation; the debug version also controls the extra checking of memory allocation.

You can set some of the variables by using environment variables, as listed below. This is an easy way to control the memory allocation, but the environment variables are checked only when your program starts.

For more information about memory allocation and debugging it, see the *Heap Analysis: Making Memory Errors a Thing of the Past* chapter of the QNX Neutrino *Programmer's Guide*.

Controlling memory allocation

`MALLOC_ARENA_SIZE`

The size of the *arena*, a chunk of memory that the memory allocator allocates and deallocates from the system. This value must be a multiple of 4 KB, and currently is limited to being less than 256 KB. Environment variable: **`MALLOC_ARENA_SIZE`**.

If *value* is 0, `malloc()` returns the current arena size; if *value* is any other value, `malloc()` returns the previous size.

`MALLOC_ARENA_CACHE_FREE_NOW`

Specify a *value* of 1 to cause the arena cache to be adjusted immediately. If **`MALLOC_ARENA_CACHE_FREE_NOW`** is 0, the changes made to the cache

parameters take effect whenever memory is subsequently released to the cache.

MALLOC_ARENA_CACHE_MAXBLK

The number of cached arena blocks. Environment variable:

MALLOC_ARENA_CACHE_MAXBLK.

MALLOC_ARENA_CACHE_MAXSZ

The total size of the cached arena blocks. Environment variable:

MALLOC_ARENA_CACHE_MAXSZ.

MALLOC_FREE_CHECK

Specify a *value* of 1 to tell the library to check to see if a block that's being passed to [free\(\)](#) (p. 717) has already been freed. This option is available only when you're using `librcheck` (the regular version or the debug version of `librcheck.so`). If you're using `librcheck.so`, and you enable this option, then if you try to free a block that's already freed, the library calls [assert\(\)](#) (p. 161).

MALLOC_MEMORY_HOLD

Specify a *value* of 1 to tell the allocator to never release memory back to the system from its arena cache. Environment variable:

MALLOC_MEMORY_HOLD.

MALLOC_STATS

Get statistics about the current state of the heap allocator. The *value* argument is a pointer to a `struct malloc_stats` (defined in `<malloc.h>`). You need to allocate the memory for the structure. The typical usage is as follows:

```
struct malloc_stats ms;
...
mallopt(MALLOC_STATS, &ms);
```

MALLOC_VERIFY_ON

Specify a *value* of 1 to enable additional checking on operations involving the heap. When this option is enabled, operations through the allocator go through additional verification. If any of the verifications of the state of the heap fail, the allocator calls [assert\(\)](#) (p. 161). This command is available only when you're using `librcheck` (the regular version or the debug version of `librcheck.so`).

Controlling extra checking

If you're using the debug version of *mallopt()*, you can use the following options to control additional checks in the library:

MALLOC_CHKACCESS

Turn on (or off) boundary checking for memory and string operations.

Environment variable: **MALLOC_CHKACCESS**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

MALLOC_CHKALLOC

Turn on (or off) the argument checking for **alloc()* and *free()* functions.

Environment variable: **MALLOC_CHKALLOC**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

MALLOC_CKBOUNDS

Turn on (or off) fill-area boundary checking.

Environment variable: **MALLOC_CKBOUNDS**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

MALLOC_CHKCHAIN

Enable (or disable) full-chain checking. For each of the above options, an integer argument value of one indicates that the given type of checking should be enabled from that point onward.

Environment variable: **MALLOC_CHKCHAIN**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

MALLOC_DUMP_LEAKS

Enable leak detection on process exit if it's set. This option controls additional checks in the library and its purpose is only for the environment variable; not for the API constant.

Environment variable: ***MALLOC_DUMP_LEAKS***.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

MALLOC_EVENTBTDEPTH

Set the size of the backtrace to report for memory problems. The default size is 5.

Environment variable: ***MALLOC_EVENTBTDEPTH***.

MALLOC_START_TRACING

Enable (disable) the automatic tracing of memory when the application starts. If tracing isn't enabled when application starts, at this point it can be enabled by using only the API, or the control thread interface (see the “Create control thread” option in the Analyzing a running program topic in the Analyzing Memory Usage and Finding Errors chapter of the IDE *User's Guide*.)

Environment variable: ***MALLOC_START_TRACING***.

The *value* argument can be:

- zero to disable the checking
- 1 to enable it

MALLOC_TRACEMIN

Set the integer value to determine the number of the minimum size of allocations to track; only trace the allocations greater than or equal to the minimum number specified (in bytes).

Environment variable: ***MALLOC_TRACEMIN***.

MALLOC_TRACEMAX

Set the integer value to determine the number of the maximum size of allocations to track; only trace the allocations to that less than or equal to the maximum number specified (in bytes). Specify 0 to indicate an unlimited number of allocations.

Environment variable: ***MALLOC_TRACMAX***.

MALLOC_TRACEBTDEPTH

Set the size of the backtrace stored for allocations. The default size is 5.

Environment variable: **MALLOC_TRACEBTDEPTH**.

MALLOC_TRACING

Turn on (or off) the tracing of allocations.

Environment variable: **MALLOC_TRACING**.

MALLOC_VERIFY

Perform a chain check. If an error is found, perform error handling. The *value* argument is currently ignored; pass 1 for it.

MALLOC_FATAL

Specify the malloc fatal handler.

Environment variable: **MALLOC_FATAL**.

Use one of the following for the *value* arguments:

Symbolic name	Value	Description
M_HANDLE_IGNORE	0	Cause the program to dump a core file.
M_HANDLE_ABORT	1	Terminate execution with a call to <i>abort()</i>
M_HANDLE_EXIT	2	Exit immediately
M_HANDLE_CORE	3	Cause the program to dump a core file
M_HANDLE_SIGNAL	4	Stop the program when this error occurs

MALLOC_WARN

Specify the malloc warning handler. The values are similar to **MALLOC_FATAL**; see above.

Environment variable: **MALLOC_WARN**.

For details, see “Controlling the level of checking” in the Heap Analysis: Making Memory Errors a Thing of the Past chapter of the QNX Neutrino *Programmer's Guide*.

Returns:

0 on success, or -1 if an error occurs ([errno](#) (p. 539) is set).

For MALLOC_ARENA_SIZE, if *value* is 0, *malloc()* returns the current arena size; if *value* is any other value, *malloc()* returns the previous size.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

max()

Return the greater of two numbers

Synopsis:

```
#include <stdlib.h>

#define max(a,b) ...
```

Arguments:

a,b

The numbers that you want to get the greater of.

Description:

The *max()* function returns the greater of two values.



The *max()* function is for C programs only. For C++ programs, use the `__max()` macro.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int a;

    a = max( 1, 10 );
    printf( "The value is: %d\n", a );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

mblen()

Count the bytes in a multibyte character

Synopsis:

```
#include <stdlib.h>

int mblen( const char * s,
           size_t n );
```

Arguments:

s

NULL (see below), or a pointer to a multibyte character.

n

The maximum number of bytes that you want to count.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mblen()* function counts the number of bytes in the multibyte character pointed to by *s*, to a maximum of *n* bytes.

The [mbrlen\(\)](#) (p. 1585) function is a restartable version of *mblen()*.

Returns:

- If *s* is NULL, *mblen()* determines whether or not the character encoding is state-dependent:

0

The *mblen()* function uses locale-specific multibyte character encoding that's not state-dependent.

0

Character is state-dependent.

- If *s* isn't NULL:

0

s points to the null character.

-1

The next *n* bytes don't form a valid multibyte character.

> 0

The number of bytes that comprise the multibyte character (if the next *n* or fewer bytes form a valid multibyte character).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int len;
    char *mbs = "string";

    printf( "Character encodings do " );
    if( !mblen( NULL, 0 ) ) {
        printf( "not " );
    }
    printf( "have state-dependent \nencoding.\n" );

    len = mblen( "string", 6 );
    if( len != -1 ) {
        mbs[len] = '\0';
        printf( "Multibyte char '%s'(%d)\n", mbs, len );
    }

    return EXIT_SUCCESS;
}
```

This produces the output:

```
Character encodings do not have state-dependent
encoding.
Multibyte char 's'(1)
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mbrlen()

Count the bytes in a multibyte character (restartable)

Synopsis:

```
#include <wchar.h>

size_t mbrlen( const char * s,
               size_t n,
               mbstate_t * ps);
```

Arguments:

s

A pointer to a multibyte character.

n

The maximum number of bytes that you want to count.

ps

An internal pointer that lets *mbrlen()* be a restartable version of [mblen\(\)](#) (p. 1583); if *ps* is NULL, *mbrlen()* uses its own internal variable.

You can call [mbsinit\(\)](#) (p. 1589) to determine the status of this variable.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mbrlen()* function counts the bytes in the multibyte character pointed to by *s*, to a maximum of *n* bytes.

Returns:

(size_t)-2

The resulting conversion state indicates an incomplete multibyte character after all *n* characters were converted.

(size_t)-1

The function detected an encoding error before completing the next multibyte character, in which case the function [errno](#) (p. 539) to EILSEQ and leaves the resulting conversion state undefined.

0

The next completed character is a null character, in which case the resulting conversion state is the initial conversion state.

x

The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that x bytes have been converted.

Errors:

EILSEQ

Invalid character sequence.

EINVAL

The *ps* argument points to an invalid object.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mbrtowc()

Convert a multibyte character into a wide character (restartable)

Synopsis:

```
#include <wchar.h>

size_t mbrtowc( wchar_t * pwc,
                const char * s,
                size_t n,
                mbstate_t * ps );
```

Arguments:

pwc

A pointer to a `wchar_t` object where the function can store the wide character.

s

A pointer to the multibyte character that you want to convert.

n

The maximum number of bytes in the multibyte character to convert.

ps

An internal pointer that lets *mbrtowc()* be a restartable version of [mbtowc\(\)](#) (p. 1595); if *ps* is NULL, *mbrtowc()* uses its own internal variable.

You can call [mbsinit\(\)](#) (p. 1589) to determine the status of this variable.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mbrtowc()* function converts single multibyte characters pointed to by *s* into wide characters pointed to by *pwc*, to a maximum of *n* bytes (not characters).

This function is affected by `LC_TYPE`.

Returns:

(size_t)-2

After converting all *n* characters, the resulting conversion state indicates an incomplete multibyte character.

(size_t)-1

The function detected an encoding error before completing the next multibyte character; the function sets [errno](#) (p. 539) to EILSEQ and leaves the resulting conversion state undefined.

0

The next completed character is a null character; the resulting conversion state is the same as the initial one.

x

The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that *x* bytes have been converted.

Errors:

EILSEQ

Invalid character sequence.

EINVAL

The *ps* argument points to an invalid object.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mbsinit()

Determine the status of the conversion object used for restartable mb() functions*

Synopsis:

```
#include <wchar.h>

int mbsinit( const mbstate_t * ps );
```

Arguments:

ps

A pointer to the conversion object that you want to test.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The following functions use an object of type `mbstate_t` so that they can be restarted:

- [*mbrlen\(\)*](#) (p. 1585)
- [*mbrtowc\(\)*](#) (p. 1587)
- [*mbsrtowcs\(\)*](#) (p. 1591)
- [*mbstowcs\(\)*](#) (p. 1593)
- [*wcsrtombs\(\)*](#) (p. 3697)
- [*wcrtomb\(\)*](#) (p. 3671)

The *mbsinit()* function determines whether or not the `mbstate_t` object pointed to by *ps* describes an initial conversion state.



If the object doesn't describe an initial conversion state, it isn't safe for you to use it in one of the above functions, other than the one you've already used it in.

Returns:

A nonzero value if *ps* is NULL or **ps* describes an initial conversion state; otherwise zero.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mbsrtowcs()

Convert a multibyte-character string into a wide-character string (restartable)

Synopsis:

```
#include <wchar.h>

size_t mbsrtowcs( wchar_t * dst,
                  const char ** src,
                  size_t n,
                  mbstate_t * ps );
```

Arguments:

dst

A pointer to a buffer where the function can store the wide-character string.

src

The string of multibyte characters that you want to convert.

n

The maximum number of wide characters to store in the buffer that *dst* points to.

ps

An internal pointer that lets *mbsrtowcs()* be a restartable version of [mbstowcs\(\)](#) (p. 1593); if *ps* is NULL, *mbsrtowcs()* uses its own internal variable.

You can call [mbsinit\(\)](#) (p. 1589) to determine the status of this variable.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mbsrtowcs()* function converts a string of multibyte characters pointed to by *src* into the corresponding wide characters pointed to by *dst*, to a maximum of *n* wide characters, including the terminating NULL character.

The function converts each character as if by a call to [mbtowc\(\)](#) (p. 1595) and stops early if:

- a sequence of bytes doesn't conform to a valid character

Or:

- converting the next character would exceed the limit of *n* wide characters

This function is affected by LC_TYPE.

Returns:

(size_t)-1

Failure; invalid wide-character code.

x

Success; the number of total characters successfully converted, not including the terminating NULL character.

Errors:

EILSEQ

Invalid character sequence.

EINVAL

The *ps* argument points to an invalid object.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mbstowcs()

Convert a multibyte-character string into a wide-character string

Synopsis:

```
#include <stdlib.h>

size_t mbstowcs( wchar_t * pwcs,
                  const char * s,
                  size_t n );
```

Arguments:

pwcs

A pointer to a buffer where the function can store the wide-character string.

s

The string of multibyte characters that you want to convert.

n

The maximum number of wide characters to store in the buffer that *pwcs* points to.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *mbstowcs()* function converts a sequence of multibyte characters pointed to by *s* into their corresponding wide-character codes pointed to by *pwcs*, to a maximum of *n* wide characters. It doesn't convert any multibyte characters beyond a **NULL** character.

This function is affected by `LC_TYPE`.

The [mbsrtowcs\(\)](#) (p. 1591) function is a restartable version of *mbstowcs()*.

Returns:

The number of array elements modified, not including the terminating zero code, if present, or `(size_t) -1` if an invalid multibyte character was encountered.



If the number of elements is equal to n , the array isn't terminated by a null character.

Errors:

EILSEQ

Invalid character sequence.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *wc = "string";
    wchar_t wbuffer[50];
    int i, len;

    len = mbstowcs( wbuffer, wc, 50 );
    if( len != -1 ) {
        wbuffer[len] = '\0';
        printf( "%s(%d)\n", wc, len );

        for( i = 0; i < len; i++ ) {
            printf( "%4.4x", wbuffer[i] );
        }

        printf( "\n" );
    }

    return EXIT_SUCCESS;
}
```

This produces the output:

```
string(6)
/0073/0074/0072/0069/006e/0067
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mbtowc()

Convert a multibyte character into a wide character

Synopsis:

```
#include <stdlib.h>

int mbtowc( wchar_t * pwc,
            const char * s,
            size_t n );
```

Arguments:

pwc

A pointer to a `wchar_t` object where the function can store the wide character.

s

NULL (see below), or a pointer to the multibyte character that you want to convert.

n

The maximum number of bytes in the multibyte character to convert.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mbtowc()` function converts a single multibyte character pointed to by `s` into a wide-character code pointed to by `pwc`, to a maximum of `n` bytes. The function stops early if it encounters the NULL character.

This function is affected by `LC_TYPE`.

The [mbrtowc\(\)](#) (p. 1587) function is a restartable version of `mbtowc()`.

Returns:

- If `s` is NULL:

0

The *mbtowc()* function uses UTF-8 multibyte character encoding that's not state-dependent.

0

Everything else.

- If *s* isn't NULL:

0

The *s* argument points to the NUL character.

> 0

The number of bytes that comprise the multibyte character, to a maximum of MB_CUR_MAX (if the next *n* or fewer bytes form a valid multibyte character).

-1

The next *n* bytes don't form a valid multibyte character; *errno* (p. 539) is set.

Errors:

EILSEQ

Invalid character sequence.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *wc = "string";
    wchar_t wbuffer[10];
    int i, len;

    printf( "State-dependent encoding? " );
    if( mbtowc( wbuffer, NULL, 0 ) ) {
        printf( "Yes\n" );
    } else {
        printf( "No\n" );
    }

    len = mbtowc( wbuffer, wc, 2 );
    wbuffer[len] = '\0';
    printf( "%s(%d)\n", wc, len );

    for( i = 0; i < len; i++ ) {
        printf( "%4.4x", wbuffer[i] );
    }

    printf( "\n" );
}
```



```
    return EXIT_SUCCESS;  
}
```

This produces the output:

```
State-dependent encoding? No  
string(1)  
/0073
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

mcheck()

Enable memory allocation routine consistency checks

Synopsis:

```
#include <malloc.h>

int mcheck(
    void (* abort_fn)(enum mcheck_status status));
```

Arguments:

abort_fn

A pointer to the callback function to invoke when an inconsistency in the memory-allocation routines is found, or `NULL` if you want to use the default callback routine.

The argument to the callback routine is one of the values of the `mcheck_status` enumeration described in the documentation for [mprobe\(\)](#) (p. 1697).

The default abort callback prints a message to `stderr` and aborts the application.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `mcheck()` function enables consistency checks within the memory allocation routines. When enabled, consistency checks are periodically performed on allocated memory blocks as blocks are allocated or freed. If an inconsistency is found, the `abort` callback is called with the status identifying the type of inconsistency found.



Consistency checking isn't performed on blocks that you allocated before calling `mcheck()`.

The level of checking provided depends on which version of the allocator you've linked the application with:

- C library — minimal consistency checking.

- Nondebug version of the `malloc` library — a slightly greater level of consistency checking.
- Debug version of the `malloc` library — extensive consistency checking, with tuning available through the use of the [mallopt\(\)](#) (p. 1575) function.



You can call `mcheck()` only once in a program.

Returns:

-1

Checking is already enabled.

0

Checking wasn't already enabled.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mem_offset(), mem_offset64()

Get the physical address of a virtually mapped memory block

Synopsis:

```
#include <sys/mman.h>

int mem_offset( const void * addr,
               int fd,
               size_t length,
               off_t * offset,
               size_t * contig_len );

int mem_offset64( const void * addr,
                 int fd,
                 size_t length,
                 off64_t * offset,
                 size_t * contig_len );
```

Arguments:

addr

The address of the memory block whose offset and contiguous length you want to get.

fd

This must be NOFD, or the function will fail.

length

The length of the block of memory that you want the offset for.

offset

A pointer to a location where the function can store the offset of the memory block.

contig_len

A pointer to a location where the function can store either *length* or the length of the largest contiguous block of memory that's currently mapped to the calling process starting at *addr*, whichever is smaller.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mem_offset()` and `mem_offset64()` functions set the variable pointed to by `offset` to the offset into `/dev/mem` of `addr` (i.e. its physical address). The `mem_offset64()` function is a 64-bit version of `mem_offset()`.



- If the physical address isn't a valid `off_t` value, `mem_offset()` will fail with `errno` set to `E_OVERFLOW`. This is typically the case with many ARM systems, and you should use `mem_offset64()` to get the physical address.
- These functions also cause the initial copying or zero-filling of the `MAP_PRIVATE` or `MAP_ANON` pages.
- For best performance results, you should cache the result of `mem_offset()`, rather than repeatedly call the function for a given virtual address.

To get the offset and length of a mapped typed memory block, use [posix_mem_offset\(\)](#) or [posix_mem_offset64\(\)](#) (p. 2030).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

The process hasn't mapped memory at the given address `addr`, or the address is of `MAP_LAZY` pages that aren't yet memory-resident.

ENODEV

The file descriptor `fd` isn't `NOFD`.

E_OVERFLOW

This error is specific to `mem_offset()` and is returned when the address is too large for the 32-bit `off_t`; use `mem_offset64()` instead.

Examples:

```
off64_t          offset;

if(mem_offset64(addr, NOFD, 1, &offset, 0) == -1)
{
    /* Error */
}
else
{
    /* offset contains the physical address of the memory
       mapped at addr. */
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

memalign()

Allocate aligned memory

Synopsis:

```
#include <malloc.h>

void *memalign( size_t alignment,
               size_t size );
```

Arguments:

alignment

The alignment that you want to use for the memory. This must be a multiple of `size( void *)`.

size

The amount of memory you want to allocate, in bytes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memalign()* function allocates *size* bytes aligned on a boundary specified by *alignment*.



Because the *malloc()* implementation uses signed, 32-bit integers to represent the size internally, you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, *memalign()* indicates an error of `ENOMEM`.

Returns:

A pointer to the allocated block, or `NULL` if an error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The value of *alignment* isn't a multiple of `size(void *)`, or it's greater than the value of the ***MALLOC_MAX_ALIGNMENT*** environment variable.

ENOMEM

There's insufficient memory available with the requested alignment.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

memccpy()

Copy bytes between buffers until a given byte is found

Synopsis:

```
#include <string.h>

void* memccpy( void *dest,
               const void *src,
               int c,
               size_t n );
```

Arguments:

dest

A pointer to where you want the function to copy the data.

src

A pointer to the buffer that you want to copy data from.

c

The value that you want to stop copying at.

n

The maximum number of bytes to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memccpy()* function copies bytes from *src* to *dest*, up to and including the first occurrence of the character *c* (converted into an `unsigned char`), or until *n* bytes have been copied, whichever comes first.

Returns:

A pointer to the byte in *dest* following the character *c*, if one is found and copied; otherwise, `NULL`.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char* msg = "This is the string: not copied";

int main( void )
{
    char buffer[80];

    memset( buffer, '\0', 80 );
    memcpy( buffer, msg, '\0', 80 );

    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}
```

produces the output:

This is the string:

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memchr()

Find the first occurrence of a character in a buffer

Synopsis:

```
#include <string.h>

void* memchr(void* buf,
             int ch,
             size_t length );
```

Arguments:

buf

The buffer that you want to search.

ch

The character that you're looking for.

length

The number of bytes to search in the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memchr()` function locates the first occurrence of `ch` (converted to an unsigned char) in the first `length` bytes of the buffer pointed to by `buf`.

Returns:

A pointer to the located character, or `NULL` if `ch` couldn't be found.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
```

```
char* where;

strcpy( buffer, "video x-rays" );

where = (char *) memchr( buffer, 'x', 6 );
if( where == NULL ) {
    printf( "'x' not found\n" );
} else {
    printf( "%s\n", where );
}

where = (char *) memchr( buffer, 'r', 9 );
if( where == NULL ) {
    printf( "'r' not found\n" );
} else {
    printf( "%s\n", where );
}

return EXIT_SUCCESS;
}
```

produces the output:

```
'x' not found
rays
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memcmp()

Compare the bytes in two buffers

Synopsis:

```
#include <string.h>

int memcmp( const void* s1,
            const void* s2,
            size_t length );
```

Arguments:

s1, s2

Pointers to the buffers that you want to compare.

length

The number of bytes that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memcmp()* function compares *length* bytes of the buffer pointed to by *s1* to the buffer pointed to by *s2*.

Returns:

< 0

The object pointed to by *s1* is less than the object pointed to by *s2*.

0

The object pointed to by *s1* is equal to the object pointed to by *s2*.

> 0

The object pointed to by *s1* is greater than the object pointed to by *s2*.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
    int retval;

    strcpy( buffer, "World" );

    retval = memcmp( buffer, "hello", 5 );
    if( retval < 0 ) {
        printf( "Less than\n" );
    } else if( retval == 0 ) {
        printf( "Equal to\n" );
    } else {
        printf( "Greater than\n" );
    }

    return EXIT_SUCCESS;
}
```

produces the output:

Less than

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memcpy()

Copy bytes from one buffer to another

Synopsis:

```
#include <string.h>

void* memcpy( void* dst,
              const void* src,
              size_t length );
```

Arguments:

dest

A pointer to where you want the function to copy the data.

src

A pointer to the buffer that you want to copy data from.

length

The number of bytes to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memcpy()* function copies *length* bytes from the buffer pointed to by *src* into the buffer pointed to by *dst*.



Copying overlapping buffers isn't guaranteed to work; use [memmove\(\)](#) (p. 1618) to copy buffers that overlap.

Returns:

A pointer to the destination buffer (that is, the value of *dst*).

Examples:

```
#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];

    memcpy( buffer, "Hello", 5 );
    buffer[5] = '\0';
    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memcpyv()

Copy data from one I/O vector to another

Synopsis:

```
#include <string.h>

size_t memcpyv( const struct iovec *dst,
                int dparts,
                int doff,
                const struct iovec *src,
                int sparts,
                int soff );
```

Arguments:

dst

An array of `iovec` structures that you want to copy the data to.

dparts

The number of entries in the *dst* array.

doff

The offset into data represented by the *dst* array to start copying to.

src

An array of `iovec` structures that you want to copy the data from.

sparts

The number of entries in the *src* array.

soff

The offset into data represented by the *src* array to start copying from.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memcpyv()* function copies data pointed to by the *src* I/O vector, starting at offset *soff*, to the *dst* I/O vector, starting at offset *doff*. The number of parts in each I/O vector is specified by *sparts* and *dparts*.

Returns:

The number of bytes copied.

Examples:

This example copies data from a two-part I/O vector, starting at offset 5, into a three-part vector, starting at offset 3:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void print_buffer (char * buffer, char * buffer_name,
                  int buffer_len)
{
    int i;

    printf ("  %s: '", buffer_name);
    for (i = 0; i < buffer_len; i++)
    {
        if (buffer[i] == 0)
            printf ("_");
        else
            printf ("%c", buffer[i]);
    }
    printf ("'\n");
}

int main( void )
{
    char msg1a[10]="1234567890",
        msg1b[20]="abcdefghijklmnop",
        msg2a[10], msg2b[10], msg2c[20];

    iov_t dest[3], source[2];
    int dparts=3, doffset=3, sparts=2, soffset=5;
    size_t nbytes;

    SETIOV (source + 0, &msg1a, sizeof (msg1a));
    SETIOV (source + 1, &msg1b, sizeof (msg1b));

    printf ("Original data:\n");
    print_buffer (msg1a, "msg1a", sizeof (msg1a));
    print_buffer (msg1b, "msg1b", sizeof (msg1b));

    SETIOV (dest + 0, &msg2a, sizeof (msg2a));
    SETIOV (dest + 1, &msg2b, sizeof (msg2b));
    SETIOV (dest + 2, &msg2c, sizeof (msg2c));
    memset (msg2a, 0, sizeof (msg2a));
    memset (msg2b, 0, sizeof (msg2b));
    memset (msg2c, 0, sizeof (msg2c));

    nbytes = memcpyv ( dest, dparts, doffset,
                      source, sparts, soffset );
    printf ( "\nCopied %d bytes:\n", nbytes );
    print_buffer (msg2a, "msg2a", sizeof (msg2a));
    print_buffer (msg2b, "msg2b", sizeof (msg2b));
    print_buffer (msg2c, "msg2c", sizeof (msg2c));

    return EXIT_SUCCESS;
}
```

This program produces the following output:

```
Original data:
msg1a: '1234567890'
msg1b: 'abcdefghijklmnop____'

Copied 25 bytes:
msg2a: '____67890ab'
msg2b: 'cdefghijkl'
msg2c: 'mnop_____'
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memicmp()

Compare two buffers, ignoring case

Synopsis:

```
#include <string.h>

int memicmp( const void* s1,
             const void* s2,
             size_t length );
```

Arguments:

s1, s2

Pointers to the buffers that you want to compare.

length

The number of bytes that you want to compare.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *memicmp()* function compares (case insensitive) *length* bytes of the buffer pointed to by *s1* with those of the buffer pointed to by *s2*.

Returns:

0

The object pointed to by *s1* is the same as the object pointed to by *s2*.

Less than 0

The object pointed to by *s1* is less than the object pointed to by *s2*.

Greater than 0

The object pointed to by *s1* is greater than the object pointed to by *s2*.

Examples:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
    int retval;

    strcpy( buffer, "World" );

    retval = memicmp( buffer, "hello", 5 );
    if( retval < 0 ) {
        printf( "Less than\n" );
    } else if( retval == 0 ) {
        printf( "Equal\n" );
    } else {
        printf( "Greater than\n" );
    }

    return EXIT_SUCCESS;
}

```

produces the output:

```
Less than
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memmove()

Copy bytes from one buffer to another, handling overlapping memory correctly

Synopsis:

```
#include <string.h>

void* memmove( void* dst,
               const void* src,
               size_t length );
```

Arguments:

dest

A pointer to where you want the function to copy the data.

src

A pointer to the buffer that you want to copy data from.

length

The number of bytes to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memmove()* function copies *length* bytes from the buffer pointed to by *src* to the buffer pointed to by *dst*. Copying of overlapping regions is handled safely. Use [memcpy\(\)](#) (p. 1611) for greater speed when copying buffers that don't overlap.

Returns:

A pointer to the destination buffer (that is, the value of *dst*).

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
```

```

char buffer[80];

strcpy( buffer, "World");
memmove( buffer+1, buffer, 79 );
printf ("%s\n", buffer);

return EXIT_SUCCESS;
}

```

produces the output:

```

wWorld

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memset()

Set memory to a given value

Synopsis:

```
#include <string.h>

void* memset( void* dst,
              int c,
              size_t length );
```

Arguments:

dst

A pointer to the memory that you want to set.

c

The value that you want to store in each byte.

length

The number of bytes to set.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memset()* function fills *length* bytes starting at *dst* with the value *c*.



The compiler might optimize out calls to *memset* if it appears that the memory isn't used again in the calling function. For a secure routine that forces the memory to be set, call [memset_s\(\)](#) (p. 1622).

Returns:

A pointer to the destination buffer (that is, the value of *dst*).

Examples:

```
#include <string.h>
```



```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    char buffer[80];

    memset( buffer, '=', 80 );
    buffer[79] = '\\0';

    puts( buffer );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

memset_s()

Securely set memory to a given value

Synopsis:

```
#include <string.h>

errno_t memset_s( void *s,
                  rsize_t smax,
                  int c,
                  size_t n );
```

Arguments:

s

A pointer to the memory that you want to set. This must not be **NULL**.

smax

The maximum size of the block of memory.

c

The value that you want to store in each byte.

n

The number of bytes to set. This value must not be greater than the value of *smax*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memset_s()* function is a secure version of [memset\(\)](#) (p. 1620):

- If the arguments are all valid, *memset_s()* copies the value of *c* (converted to an `unsigned char`) into each of the first *n* characters of the object pointed to by *s*.
- If there's a runtime-constraint violation, then if *s* isn't **NULL**, *memset_s()* stores the value of *c* (converted to an `unsigned char`) into each of the first *smax* characters of the object pointed to by *s*.

- Unlike *memset()*, any call to *memset_s()* assumes that the memory indicated by *s* may be accessible in the future and thus must contain the values indicated by *c*.

Returns:**EOK**

Success.

EINVAL

The *s* argument is NULL, or *n* is greater than *smax*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

message_attach()

Attach a message range

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int message_attach( dispatch_t * dpp,
                    message_attr_t * attr,
                    int low,
                    int high,
                    int (* func) (
                        message_context_t * ctp,
                        int code,
                        unsigned flags,
                        void * handle ),
                    void * handle );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433).

attr

A pointer to a `message_attr_t` structure that lets you specify additional requirements for the message; see “[message_attr_t structure](#) (p. 1625),” below.

low, high

The range of messages that you're interested in.



Although these arguments are of type `int`, `message_attach()` stores the values internally as `short`.

func

The function that you want to call when a message in the given range is received; see “[Handler function](#) (p. 1626),” below.

handle

An arbitrary handle that you want to associate with data for the defined message range. This handle is passed to *func*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `message_attach()` function attaches a handler to the message range defined by the message type `[low, high]` (i.e. an inclusive message range) for dispatch handle `dpp`.



It's considered a programming error to attach overlapping message or pulse ranges. Message types should be greater than `_IO_MAX` (defined in `<sys/iomsg.h>`).

When a message with a type in that range is received, `dispatch_handler()` (p. 440) calls the user-supplied function `func`. You can also use the same function with `pulse_attach()` (p. 2442). By examining `ctp->rcvid`, `func` can determine whether a pulse or message was received.

This function is responsible for doing any specific work needed to handle the message pointed to by `ctp->msg`. The `handle` passed to the function is the `handle` initially passed to `message_attach()`.

message_attr_t structure

The `attr` argument is a pointer to a `message_attr_t` structure:

```
typedef struct _message_attr {
    unsigned    flags;
    unsigned    nparts_max;
    unsigned    msg_max_size;
} message_attr_t;
```

You can use this structure to specify:

- the maximum message size to be received (the context allocated must be at least big enough to contain a message of that size)
- the maximum number of iops to reserve in the `message_context_t` structure (`attr->nparts_max`)
- various `flags`:

Currently, the following `attr->flags` are defined:

MSG_FLAG_CROSS_ENDIAN

Allow the server to receive messages from clients on machines with different native endian formats.

MSG_FLAG_DEFAULT_FUNC

Call this function if no other match is found, in this case, *low* and *high* are ignored. This overrides the default behavior of [dispatch_handler\(\)](#) (p. 440) which is to return [MsgError\(\)](#) (p. 1743) (ENOSYS) to the sender when an unknown message is received.

Handler function

The user-supplied function *func* is called when a message in the defined range is received. This function is passed the message context *ctp*, in which the message was received, the message type, and the *handle* (the one passed to *message_attach()*). Currently, the argument *flags* is reserved. Your function should return 0; other return values are reserved.

Here's a brief description of the context pointer fields:

ctp->rcvid

The receive ID of the message.

ctp->msg

A pointer to the message.

ctp->info

Data from a [_msg_info](#) (p. 1731) structure.

Returns:

Zero on success, or -1 on failure ([errno](#) (p. 539) is set).

Errors:

EINVAL

The message *code* is out of range.

ENOMEM

Insufficient memory to attach message type.

Examples:

In this example, we create a resource manager where we attach to a private message range and attach a pulse, which is then used as a timer event:

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

#define THREAD_POOL_PARAM_T    dispatch_context_t
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_func;
static resmgr_io_funcs_t        io_func;
static iofunc_attr_t            attr;

int
timer_tick(message_context_t *ctp, int code,
            unsigned flags, void *handle) {

    union sigval    value = ctp->msg->pulse.value;
    /* Do some useful work on every timer firing... */
    printf("received timer event, value %d\n", value.sival_int);
    return 0;
}

int
message_handler(message_context_t *ctp, int code,
                unsigned flags, void *handle ) {
    printf("received private message, type %d\n", code);
    return 0;
}

int
main(int argc, char **argv) {
    thread_pool_attr_t    pool_attr;
    thread_pool_t        *tpp;
    dispatch_t            *dpp;
    resmgr_attr_t        resmgr_attr;
    int                   id;
    int                   timer_id;
    struct sigevent       event;
    struct _itimr         itime;

    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr,
            "%s: Unable to allocate dispatch handle.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;
    /* We are doing resmgr and pulse-type attaches */
    pool_attr.context_alloc = dispatch_context_alloc;
    pool_attr.block_func = dispatch_block;
    pool_attr.unblock_func = dispatch_unblock;
    pool_attr.handler_func = dispatch_handler;
    pool_attr.context_free = dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;

    if((tpp = thread_pool_create(&pool_attr,
                                POOL_FLAG_EXIT_SELF)) == NULL) {
        fprintf(stderr,
            "%s: Unable to initialize thread pool.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func,
                    _RESMGR_IO_NFUNCS, &io_func);
```

```

/* You should restrict the permissions according to what you want other
processes and users to be able to do with your resource manager. */

iofunc_attr_init(&attr, S_IFNAM | 0660, 0, 0);

memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

if((id = resmgr_attach(dpp, &resmgr_attr, "/dev/mynull",
    _FTYPE_ANY, 0,
    &connect_func, &io_func, &attr)) == -1) {
    fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
    return EXIT_FAILURE;
}

/*
We want to handle our own private messages, of type
0x5000 to 0x5fff
*/
if(message_attach(dpp, NULL, 0x5000, 0x5fff,
    &message_handler, NULL) == -1) {
    fprintf(stderr,
        "Unable to attach to private message range.\n");
    return EXIT_FAILURE;
}

/* Initialize an event structure, and attach a pulse to it */
if((event.sigev_code = pulse_attach(dpp,
    MSG_FLAG_ALLOC_PULSE, 0,
    &timer_tick, NULL)) == -1) {
    fprintf(stderr, "Unable to attach timer pulse.\n");
    return EXIT_FAILURE;
}

/* Connect to our channel */
if((event.sigev_coid = message_connect(dpp,
    MSG_FLAG_SIDE_CHANNEL)) == -1) {
    fprintf(stderr, "Unable to attach to channel.\n");
    return EXIT_FAILURE;
}

event.sigev_notify = SIGEV_PULSE;
event.sigev_priority = -1;
/*
We could create several timers and use different
sigev values for each
*/
event.sigev_value.sival_int = 0;

if((timer_id = TimerCreate(CLOCK_REALTIME, &event)) == -1) {
    fprintf(stderr,
        "Unable to attach channel and connection.\n");
    return EXIT_FAILURE;
}

/* And now setup our timer to fire every second */
itime.nsec = 1000000000;
itime.interval_nsec = 1000000000;
TimerSettime(timer_id, 0, &itime, NULL);

/* Never returns */
thread_pool_start(tpp);
return EXIT_SUCCESS;
}

```

For more examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

message_connect()

Create a connection to a channel

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int message_connect( dispatch_t * dpp,
                    int flags );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433).

flags

Currently, the following flag is defined in `<sys/dispatch.h>`:

- **MSG_FLAG_SIDE_CHANNEL** — request the connection ID be returned from a different space. This ID will be greater than any valid file descriptor. Once created there's no difference in the use of the messaging primitives on these IDs.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `message_connect()` function creates a connection to the channel used by dispatch handle `dpp`. This function calls the [ConnectAttach\(\)](#) (p. 351) kernel call. To detach the connection ID, you can call [ConnectDetach\(\)](#) (p. 367).



The `message_connect()` function works only when the dispatch blocking type is receive, i.e. attaches were done for `resmgr`, `message`, or select “type” events. If no attaches were done yet, the `message_connect()` call fails, since dispatch can't determine if receive or `sigwait` blocking will be used.

Returns:

A connection ID used by the message primitives, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**EAGAIN**

All kernel connection objects are in use.

EINVAL

Dispatch *dpp* doesn't have a channel.

Examples:

```
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int flags, coid, id;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr,
            "%s: Unable to allocate dispatch context.\n",
            argv[0] );
        return EXIT_FAILURE;
    }

    id = resmgr_attach ( ... );

    ...

    if ( (coid = message_connect ( dpp, flags )) == -1 ) {
        fprintf ( stderr, "Failed to create connection \
            to channel used by dispatch.\n");
        return 1;
    }
    /* else connection to channel used by dispatch is created */

    ...
}
```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Dispatch *dpp* must block on messages.

message_detach()

Detach a message range

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int message_detach( dispatch_t * dpp,
                   int low,
                   int high,
                   int flags );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433).

low, high

The range of messages that you want to detach the handler from. This range must be the same one that you passed to [message_attach\(\)](#) (p. 1624).

flags

Reserved.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `message_detach()` function detaches the message type `[low, high]`, for dispatch handle `dpp`, that was attached with `message_attach()`.

Returns:

Zero on success. If an error occurs, -1 or the following error constant:

EINVAL

The range [*low*, *high*] doesn't match the range that you attached with [message_attach\(\)](#) (p. 1624).

Examples:

```
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int my_func( ... ) {
    ...
}

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int lo=0x2000, hi=0x2fff, flags=0;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr,
            "%s: Unable to allocate dispatch handle.\n",
            argv[0] );
        return EXIT_FAILURE;
    }

    ...

    if( message_attach( dpp, NULL, lo, hi,
        &my_func, NULL ) == -1 ) {
        fprintf( stderr,
            "%s: Failed to attach message range.\n",
            argv[0] );
        return 1;
    }

    ...

    if ( message_detach ( dpp, lo, hi, flags ) == -1 ) {
        fprintf ( stderr,
            "Failed to detach message range from %d to %d.\n",
            lo, hi );
        return 1;
    }
    /* else message was detached */

    ...
}
```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

min()*Return the lesser of two numbers***Synopsis:**

```
#include <stdlib.h>

#define min(a,b) ...
```

Arguments:***a,b***

The numbers that you want to get the lesser of.

Description:

The *min()* function returns the lesser of two values.



The *min()* function is for C programs only. For C++ programs, use the `__min()` macro.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int a;

    a = min( 1, 10 );
    printf( "The value is: %d\n", a );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

mkdir()

Create a subdirectory

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir( const char *path,
           mode_t mode );
```

Arguments:

path

The name of the directory that you want to create.

mode

The permissions for the directory, modified by the process's file-creation mask (see [umask\(\)](#) (p. 3557)).

The access permissions for the file or directory are specified as a combination of bits defined in the `<sys/stat.h>` header file. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mkdir()* function creates a new subdirectory named *path*. The *path* can be relative to the current working directory or it can be an absolute path name.



Not all filesystems support the creation of directories. For example, `/dev/shmem` (which really isn't a filesystem but looks like one) doesn't. For more information, see the Working with Filesystems chapter of the QNX Neutrino *User's Guide*.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the group ID of the parent directory (if the parent set-group ID bit is set) or to the process's effective group ID.

The newly created directory is empty.

The following bits, in addition to file permission bits, behave as follows:

S_ISGID

Files and directories created within this directory have the group ID of this directory instead of the group ID of the process creating the file or directory.

S_ISVTX

Files can be removed or renamed only if one or more of the following is true:

- The user owns the file.
- The user owns the directory.
- The file is writable by the user.
- The user is privileged.

The *mkdir()* function marks the *st_atime*, *st_ctime*, and *st_mtime* fields of the directory for update. Also, the *st_ctime* and *st_mtime* fields of the parent directory are also updated.

Returns:

0, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EACCES

Search permission is denied for a component of *path*, or write permission is denied on the parent directory of *path*.

EEXIST

The directory named by *path* already exists.

ELOOP

Too many levels of symbolic links.

EMLINK

The link count of the parent directory would exceed `LINK_MAX`.

ENAMETOOLONG

The length of *path* exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENOENT

A pathname component in the specified *path* does not exist, or *path* is an empty string.

ENOSPC

The filesystem does not contain enough space to hold the contents of the new directory or to extend the parent directory.

ENOSYS

This function is not supported for this path.

ENOTDIR

A component of *path* is not a directory.

EROFS

The parent directory resides on a read-only filesystem.

Examples:

To make a new directory called `/src` in `/hd`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    (void)mkdir( "/hd/src",
                S_IRWXU |
                S_IRGRP | S_IWRP |
                S_IROTH | S_IXOTH );

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mkfifo()

Create a FIFO special file

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo( const char* path,
            mode_t mode );
```

Arguments:

path

The pathname that you want to use for the FIFO special file.

mode

The file permission bits for the new FIFO. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mkfifo()* function creates a new FIFO special file named by the pathname pointed to by *path*. The file permission bits of the new FIFO are initialized from *mode*, modified by the process's creation mask (see *umask()*). Bits that are set in *mode* other than the file permission bits are ignored.

The FIFO owner ID is set to the process's effective user ID and the FIFO's group ID is set to the process's effective group ID.

If *mkfifo()* succeeds, the *st_mtime*, *st_ctime*, *st_atime* and *st_mtime* fields of the file are marked for update. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EACCES

A component of the path prefix denies search permission.

EEXIST

The named file already exists.

ENAMETOOLONG

The length of the *path* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENOENT

A component of the path prefix doesn't exist, or the *path* arguments points to an empty string.

ENOSPC

The directory that would contain the new file cannot be extended, or the filesystem is out of file allocation resources (that is, the disk is full).

ENOSYS

This function isn't supported for this path.

ENOTDIR

A component of the path prefix isn't a directory.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    (void)mkfifo( "hd/qnx", S_IRUSR | S_IWUSR );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mknod()

Make a new filesystem entry point

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>

int mknod( const char * path,
           mode_t mode,
           dev_t dev );
```

Arguments:

path

The pathname that you want to use for the file.

mode

A set of bits that define the file type and access permissions that you want to use. The valid file types are:

- `S_IFDIR` — create a directory.
- `S_IFIFO` — create a FIFO.

For more information, see “[Access permissions](#) (p. 3165)” in the documentation for `stat()`.

dev

Ignored.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `mknod()` makes a file, named *path*, using the file type encoded in the *mode* argument. Supported file types are directories and FIFOs.



This function is included to enhance portability with software written for Unix-compatible operating systems. For POSIX portability, use [mkdir\(\)](#) (p. 1636) or [mkfifo\(\)](#) (p. 1639) instead.

To make a directory with read-write-execute permissions for everyone, you could use the following:

```
mknod (name, S_IFDIR | 0777, 0);
```

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

A component of the *path* prefix denies search permission, or write permission is denied for the parent directory.

EEXIST

The named file already exists.

ELOOP

Too many levels of symbolic links or prefixes.

EMLINK

The link count of the parent directory would exceed LINK_MAX.

ENAMETOOLONG

The length of the *path* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT

A component of the *path* prefix doesn't exist, or the *path* arguments points to an empty string.

ENOSPC

The directory that would contain the new file cannot be extended or the filesystem is out of file allocation resources (that is, the disk is full).

ENOSYS

The *mknod()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the *path* prefix isn't a directory.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
/*
 * Create special files as a directory or FIFO
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main( int argc, char** argv )
{
    int c;
    mode_t mode = 0666;
    int ecode = 0;

    if( argc == 1 ) {
        printf( "Use: %s [-d directory] ... [-f fifo] ... \n",
            argv[0] );
        return( 0 );
    }

    while(( c = getopt( argc, argv, "d:f:" )) != -1 ) {
        switch( c ) {
            case 'd': mode = S_IFDIR | 0666; break;
            case 'f': mode = S_IFIFO | 0666; break;
        }

        if( mknod( optarg, mode, 0 ) != 0 ) {
            perror( optarg );
            ++ecode;
        }
    }

    return( ecode );
}
```

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mkstemp()

Make a unique temporary filename, and open the file

Synopsis:

```
#include <stdlib.h>

int mkstemp( char* template );
```

Arguments:

template

A template for the filename that you want to use. This template can be any file name with some number of Xs appended to it, for example /tmp/temp.XXXX.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *mkstemp()* function takes the given file name template and overwrites a portion of it to create a filename. This file name is unique and suitable for use by the application. The trailing Xs are replaced with the current process number and/or a unique letter combination. The number of unique file names *mkstemp()* can return depends on the number of Xs provided; if you specify six Xs, *mkstemp()* tests roughly 26^6 combinations.

The *mkstemp()* function (unlike [mktemp\(\)](#) (p. 1647)) creates the template file, mode 0600 (i.e. read-write for the owner), returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

Returns:

The file descriptor of the temporary file, or -1 if no suitable file could be created; [errno](#) (p. 539) is set.

Errors:

ENOTDIR

The pathname portion of the template isn't an existing directory.

This function may also set *errno* to any value specified by [open\(\)](#) (p. 1905) and [stat\(\)](#) (p. 3164).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

It's possible to run out of letters. The *mkstemp()* function doesn't check to determine whether the file name part of template exceeds the maximum allowable filename length.

For portability with X/Open standards prior to XPG4v2, use [tmpfile\(\)](#) (p. 3478) instead.

mktemp()

Make a unique temporary filename

Synopsis:

```
#include <stdlib.h>

char* mktemp( char* template );
```

Arguments:

template

A template for the filename that you want to use. This template can be any file name with some number of Xs appended to it, for example /tmp/temp.XXXX.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *mktemp()* function takes the given file name template and overwrites a portion of it to create a filename. This file name is unique and suitable for use by the application. The trailing Xs are replaced with the current process number and/or a unique letter combination. The number of unique file names *mktemp()* can return depends on the number of Xs provided; if you specify six Xs, *mktemp()* tests roughly 26^6 combinations.

The [mkstemp\(\)](#) (p. 1645) function (unlike this function) creates the template file, mode 0600 (i.e. read-write for the owner), returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

Returns:

A pointer to the template, or NULL on failure; [errno](#) (p. 539) is set.

Errors:

ENOTDIR

The pathname portion of the template isn't an existing directory.

This function may also set *errno* to any value specified by [stat\(\)](#) (p. 3164).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

In general, avoid using *mktemp()*, because a hostile process can exploit a race condition in the time between the generation of a temporary filename by *mktemp()* and the invoker's use of the temporary name. Use [mkstemp\(\)](#) (p. 1645) instead.

This function can create only 26 unique file names per thread for each unique template.

mktime()

Convert local time to calendar time

Synopsis:

```
#include <time.h>

time_t mktime( struct tm* timeptr );
```

Arguments:

timeptr

A pointer to a [tm](#) (p. 3476) structure that contains the local time that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mktime()* function converts the local time information in the `struct tm` (p. 3476) specified by *timeptr* into a calendar time (Coordinated Universal Time) with the same encoding used by the [time\(\)](#) (p. 3425) function.

The original values of the *tm_sec*, *tm_min*, *tm_hour*, *tm_mday* and *tm_mon* fields aren't restricted to the ranges described for `struct tm` (p. 3476). If these fields aren't in their proper ranges, they're adjusted so that they are. Values for the fields *tm_wday* and *tm_yday* are computed after all the other fields have been adjusted.

The original value of *tm_isdst* is interpreted as follows:

< 0

This field is computed as well.

0

Daylight savings time isn't in effect.

> 0

Daylight savings time is in effect.

Whenever `mktime()` is called, the [tzset\(\)](#) (p. 3547) function is also called.

Returns:

The converted calendar time, or -1 if `mktime()` can't convert it.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

static const char *week_day[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};

int main( void )
{
    struct tm new_year;
    time_t t;

    new_year.tm_year = 2001 - 1900;
    new_year.tm_mon = 0;
    new_year.tm_mday = 1;
    new_year.tm_hour = 0;
    new_year.tm_min = 0;
    new_year.tm_sec = 0;
    new_year.tm_isdst = 0;

    t = mktime( &new_year );
    if ( t == (time_t)-1 )
        printf("No conversion possible.\n");
    else
        printf( "The 21st century began on a %s.\n",
            week_day[ new_year.tm_wday ] );

    return EXIT_SUCCESS;
}
```

produces the output:

The 21st century began on a Monday.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mlock()

Lock a range of process address space in physical memory

Synopsis:

```
#include <sys/mman.h>

int mlock(const void * addr,
          size_t len);
```

Arguments:

addr

The starting address for the range of process address space.

len

The amount of the memory to lock, in bytes. There's no limit on the amount of memory that a process may lock, other than the amount of physical memory in the system.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mlock()` function locks a range of process address space starting at address `addr` and continuing for length `len`. The `addr` must be a multiple of `PAGESIZE`, which depends on the target platform.

The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.



In order to lock memory, your process must have the `PROCmgr_AID_MEM_LOCK` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The successful call to `mlock()` function ensures that the pages are *memory-resident* (i.e. the addresses always reside in physical memory). For more information, see “Locking memory” in the Process Manager chapter of the *System Architecture* guide.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ENOMEM

- Some or all of the address range specified by the *addr* and *len* arguments doesn't correspond to valid mapped pages in the address space of the process.
- Locking the pages mapped by the specified range would exceed an implementation-defined limit on the amount of memory that the process may lock. This implementation-defined limit is set by `RLIMIT_MEMLOCK` *rlimit*.

EAGAIN

Some or all of the memory identified by the operation couldn't be locked when the call was made.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 MLR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mlockall()

Lock all of a process's address space

Synopsis:

```
#include <sys/mman.h>

int mlockall(int flags);
```

Arguments:

flags

Flags that indicate which pages to lock; one or more of the following bits:

MCL_CURRENT

Lock the pages currently mapped into the address space of the process.

MCL_FUTURE

Lock the pages that become mapped into the address space of the process in the future, when the mappings are established.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *mlockall()* function causes all of the pages mapped by the address space of a process to be locked and made memory-resident until unlocked, or the process exits or executes another process. The pages that are locked depend on the *flags* argument.



The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.

Memory-resident is a term used to indicate that the addresses always reside in physical memory. For more information, see “Locking memory” in the Process Manager chapter of the *System Architecture* guide.

In order to lock pages, your process must have the `PROC_MGR_AID_MEM_LOCK` ability enabled. For more information, see [proc_mgr_ability\(\)](#) (p. 2141).

Follow either of the following approaches when attempting to lock pages:

Tightly controlled approach

Lock the pages one by one, by calling `mmap()` and then `mlock()`.

Global approach

Lock all pages at the same time, by doing one of the following:

- Call `mlockall(MCL_FUTURE)`, followed by one or more calls to `mmap()`.
- Optionally call `mmap()`, and then call `mlockall(MCL_CURRENT)`.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

Some or all of the memory identified by the operation couldn't be locked when the call was made.

EINVAL

The *flags* argument is zero.

ENOMEM

Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock. lock. This implementation-defined limit is set by `RLIMIT_MEMLOCK rlimit`.

EPERM

The calling process doesn't have the required permission; see [proc_mgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 ML](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mmap(), mmap64()

Map a memory region into a process's address space

Synopsis:

```
#include <sys/mman.h>

void * mmap( void * addr,
             size_t len,
             int prot,
             int flags,
             int fildes,
             off_t off );

void * mmap64( void * addr,
               size_t len,
               int prot,
               int flags,
               int fildes,
               off64_t off );
```

Arguments:

addr

NULL, or a pointer to where you want the object to be mapped in the calling process's address space.

len

The number of bytes to map into the caller's address space. It can't be 0.

prot

The access capabilities that you want to use for the memory region being mapped. You can combine at least the following protection bits, as defined in `<sys/mman.h>`:

- **PROT_EXEC** — the region can be executed.

In order to successfully use this flag:



- Your process must have the **PROCmgr_AID_PROT_EXEC** ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- If your process has any privileged abilities enabled, then any memory-mapped files in the region must be from a trusted

filesystem. For more information, see `pathtrust` in the *Utilities Reference*.

-
- `PROT_NOCACHE` — disable caching of the region (e.g. so it can be used to access dual-ported memory).
 - `PROT_NONE` — the region can't be accessed.
 - `PROT_READ` — the region can be read.
 - `PROT_WRITE` — the region can be written.

flags

Flags that specify further information about handling the mapped region. POSIX defines the following:

- `MAP_PRIVATE`
- `MAP_SHARED`
- `MAP_FIXED`

The following are Unix or QNX Neutrino extensions:

- `MAP_ANON`
- `MAP_BELOW`
- `MAP_BELOW16M`
- `MAP_ELF`
- `MAP_FILE` — defined for compatibility; has no effect.
- `MAP_LAZY`
- `MAP_NOINIT`
- `MAP_RENAME` — defined for compatibility; has no effect.
- `MAP_NORESERVE` — defined for compatibility; has no effect.
- `MAP_NOSYNCFILE`
- `MAP_NOX64K`
- `MAP_PHYS`
- `MAP_STACK`

For more information, see below.

fildev

The file descriptor for a file, shared memory object, or typed memory object. If you're mapping anonymous or physical memory, this argument must be `NOFD`.

off

The offset into the file or memory object of the region that you want to start mapping, or a physical address (e.g. for mapping a device's registers in a resource manager).

If you want to map a device's physical memory, use [mmap_device_memory\(\)](#) (p. 1668) instead of [mmap\(\)](#).



On x86, if a device's registers are memory-mapped, you can use [mmap_device_memory\(\)](#) (p. 1668) to access them. Otherwise you can use [mmap_device_io\(\)](#) (p. 1666) to gain access to the device, and routines such as [in8\(\)](#) (p. 1059) and [out8\(\)](#) (p. 1923) to access them. On architectures other than the x86, both methods are equivalent.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [mmap\(\)](#) and [mmap64\(\)](#) functions map a region within the object specified by *filides*, beginning at *off* and continuing for *len*, into the caller's address space and returns the location. The [mmap64\(\)](#) function is a 64-bit version of [mmap\(\)](#). The object that you map from can be one of the following:

- a file, opened with [open\(\)](#) (p. 1905)
- a shared memory object, opened with [shm_open\(\)](#) (p. 2935)
- a typed memory object, opened with [posix_typed_mem_open\(\)](#) (p. 2124)
- physical memory — specify `NOFD` for *filides*

If *filides* isn't `NOFD`, you must have opened the file descriptor for reading, no matter what value you specify for *prot*; write access is also required for `PROT_WRITE` if you haven't specified `MAP_PRIVATE`.

The mapping is as shown below:

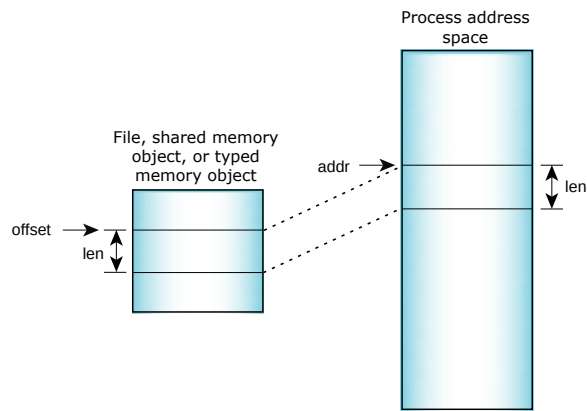


Figure 1: Mapping memory with *mmap()*.

Typically, you don't need to use *addr*; you can just pass NULL instead.

There are two parts to the *flags* parameter. The first part is a type (masked by the MAP_TYPE bits), which is maintained across a *fork()* (p. 685). You must specify one of the following types:

MAP_PRIVATE

The mapping is private to the calling process; changes aren't propagated back to the underlying object. For this type of mapping, *mmap()* allocates system RAM and copies the current object.

MAP_SHARED

The mapping may be shared by many processes; changes *are* propagated back to the underlying object.

You can OR the following flags into the above type to further specify the mapping:

MAP_ANON

Map anonymous memory that isn't associated with a specific file descriptor; the *fil*des parameter must be NOFD. The *mmap()* function allocates the memory and fills it with zeros, unless you've also specified MAP_NOINIT (see below). This is equivalent to opening /dev/zero.

MAP_ANON is most commonly used with MAP_PRIVATE, but you can use it with MAP_SHARED to create a shared memory area for forked applications.

MAP_BELOW

Map below the given *addr*, if possible.

MAP_BELOW16M

Used with `MAP_PHYS` | `MAP_ANON`. The allocated memory area resides in physical memory below 16 MB. This is important for using DMA with ISA bus devices.

MAP_ELF

The memory is an ELF object. This flag is used by the program loader.

MAP_FILE

Map a regular file. This flag is required on some operating systems, but is optional on QNX Neutrino.



You can have more than one writeable mapping to a file. If you access the file via file descriptors as well as via the memory mapping, you'll have to be careful to keep the file's contents coherent. For example, you should specify `O_SYNC` when you [open\(\)](#) (p. 1905) the file.

MAP_FIXED

Map the object to the address specified by *addr*, or the function fails. If this area is already mapped, the call changes the existing mapping of the area. A memory area being mapped with `MAP_FIXED` is first unmapped by the system using the same memory area. See [munmap\(\)](#) (p. 1832) for details.



- In order to use `MAP_FIXED`, your process must have the `PROCmgr_AID_MAP_FIXED` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- Use `MAP_FIXED` with caution. Not all memory models support it. In general, you should assume that you can `MAP_FIXED` only at an address (and size) that a call to `mmap()` without `MAP_FIXED` returned.

If *addr* isn't `NULL`, and you don't set `MAP_FIXED`, then the value of *addr* is taken as a hint as to where to map the object in the calling process's address space. The mapped area won't overlay any current mapped areas.

MAP_LAZY

Delay acquiring system memory, and copying or zero-filling the `MAP_PRIVATE` or `MAP_ANON` pages, until an access to the area has occurred. If you set this flag, and there's no system memory at the time of the access, the thread gets a `SIGBUS` with a code of `BUS_ADRERR`. This flag is a hint to the memory manager.

For anonymous shared memory objects (those created via *mmap()* with `MAP_ANON` | `MAP_SHARED` and a file descriptor of -1), a `MAP_LAZY` flag implicitly sets the `SHMCTL_LAZY` flag on the object (see [shm_ctl\(\)](#) (p. 2925)).

MAP_NOINIT

(QNX Neutrino Core OS 6.3.2 and later) Relax the POSIX requirement that the memory be zeroed. The physical memory being used for this allocation must have been previously freed with `UNMAP_INIT_OPTIONAL` for this flag to have any effect.

MAP_RENAME

Defined for compatibility, but has no effect.

MAP_NORESERVE

Defined for compatibility, but has no effect.

MAP_NOSYNCFILE

Don't update the underlying file.



If the same region in a file is mapped twice, once with `MAP_NOSYNCFILE` and once without, the memory manager might not be able to tell whether a change was made through the `MAP_NOSYNCFILE` mapping or not, and thus write out changes that weren't intended.

MAP_NOX64K

(Useful on x86 only). Used with `MAP_PHYS` | `MAP_ANON`. Prevent the allocated memory area from crossing a 64 KB boundary. This may be important to some DMA devices. If more than 64 KB is requested, the area begins on a 64 KB boundary.

MAP_PHYS

Physical memory is required. The *fildev* parameter must be `NOFD`. When you use this flag without `MAP_ANON`, the offset specifies the exact physical address to map (e.g. for video frame buffers), and is equivalent to opening `/dev/mem`.

If you use `MAP_PHYS` with `MAP_ANON`, *mmap()* allocates physically contiguous memory and ignores the offset. You can use `MAP_NOX64K` and `MAP_BELOW16M` to further define the `MAP_ANON` allocated memory (useful on x86 only).



- In order to use `MAP_PHYS`, your process must have the `PROCMGR_AID_MEM_PHYS` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
 - You should use [mmap_device_memory\(\)](#) (p. 1668) instead of `MAP_PHYS`, unless you're allocating physically contiguous memory.
-

MAP_STACK

This flag tells the memory allocator what the `MAP_ANON` memory will be used for. It's only a hint.

The following flags are defined in `<sys/mman.h>`, but you shouldn't use them:

MAP_PRIVATEANON

Deprecated; use `MAP_PRIVATE` | `MAP_ANON` instead.

MAP_SYSRAM

An output-only status flag for the `DCMD_PROC_MAPINFO` and `DCMD_PROC_PAGEDATA devctl()` commands. For more information, see “Controlling processes via the `/proc` filesystem” in the Processes chapter of the QNX Neutrino *Programmer's Guide*.



Specifying the `MAP_SYSRAM` bit in a call to `mmap()` will result in an error of `EINVAL`.

If *fildevs* represents a typed memory object opened with either the `POSIX_TYPED_MEM_ALLOCATE` or `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag (see [posix_typed_mem_open\(\)](#) (p. 2124)), and there are enough resources available, `mmap()` maps *len* bytes allocated from the corresponding typed memory object that weren't previously allocated to any process in any processor that may access that typed memory object. If there aren't enough resources available, `mmap()` fails.

If *fildevs* represents a typed memory object opened with the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the allocated bytes are contiguous within the typed memory object. If the typed memory object was opened with `POSIX_TYPED_MEM_ALLOCATE`, the allocated bytes may be composed of noncontiguous fragments within the typed memory object. If the typed memory object was opened with neither of these flags, *len* bytes starting at the given offset within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same offset and length and using file descriptors that refer to the same memory pool (either

from the same port or from a different port), both processes map the same region of storage.

Returns:

The address of the mapped-in object, or MAP_FAILED if an error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

One of the following occurred:

- The file descriptor in *fil-des* isn't open for reading.
- You specified PROT_WRITE and MAP_SHARED, and *fil-des* isn't open for writing.
- You specified PROT_EXEC for a memory-mapped file mapping, the file doesn't have execute permission for the client process, and *procnto* was started with the -mX option.

EAGAIN

The mapping couldn't be locked in memory, if required by [mlockall\(\)](#) (p. 1653), because of a lack of resources.

EBADF

Invalid file descriptor, *fil-des*.

EBUSY

The resource you're trying to map is busy.

EINVAL

The type specified in *flags* is invalid, *len* is 0, or you specified MAP_FIXED and *addr* is invalid.

EMFILE

The number of mapped regions has reached the maximum limit; see the RLIMIT_AS and RLIMIT_DATA resources for [setrlimit\(\)](#) (p. 2899).

ENODEV

The *fil-des* argument refers to an object for which *mmap()* is meaningless (e.g. a terminal).

ENOMEM

One of the following occurred:

- You specified `MAP_FIXED`, and the address range requested is outside of the allowed process address range, or there wasn't enough memory to satisfy the request.
- The mapping couldn't be locked in memory, if required by [`mlockall\(\)`](#) (p. 1653), because there wasn't enough memory.
- There aren't enough unallocated resources remaining in the typed memory object associated with the file descriptor.

ENXIO

One of the following occurred:

- Addresses in the range `[off, off+len)` are invalid for the object specified by *files*.
- You specified `MAP_FIXED`, and *addr*, *len*, and *off* are invalid for the requested object.
- The file descriptor refers to a typed memory object that the calling process can't access.

EOVERFLOW

The file is a regular file, and the offset plus the length exceeds the maximum offset.

EPERM

The calling process doesn't have the required permission (see [`procmgr\_ability\(\)`](#) (p. 2141)), or it attempted to set `PROT_EXEC` for a region of memory covered by an untrusted memory-mapped file.

Examples:

Open a shared memory object and share it with other processes:

```
fd = shm_open( "/datapoints", O_RDWR, 0777 );
addr = mmap( 0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0 );
```

Allocate a physically contiguous DMA buffer for a bus-mastering PCI network card:

```
addr = mmap( 0,
             262144,
             PROT_READ|PROT_WRITE|PROT_NOCACHE,
             MAP_PHYS|MAP_ANON,
             NOFD,
             0 );
```

Map a file into memory, change the memory, and then verify that the file's contents have been updated:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
```

```

#include <unistd.h>
#include <string.h>

#define TESTSTRING "AAAAAAAAA"

int main(int argc, char *argv[]) {
    char buffer[80], filename[200] = "/tmp/try_it";
    int fd, file_size, ret, size_written, size_read;
    void *addr;

    /* Write the test string into the file. */
    unlink( filename);
    fd = open( filename, O_CREAT|O_RDWR , 0777 );
    if( fd == -1 ) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    size_written = write( fd, TESTSTRING, sizeof (TESTSTRING) );
    if ( size_written == -1 ){
        perror("write");
        exit(0);
    }
    printf( "Wrote %d bytes into file %s\n", size_written, filename );

    lseek( fd, 0L, SEEK_SET );
    file_size = lseek( fd, 0L, SEEK_END );
    printf( "Size of file = %d bytes\n", file_size );

    /* Map the file into memory. */
    addr = mmap( 0, file_size, PROT_READ | PROT_WRITE, MAP_SHARED,
                fd, 0 );
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    /* Change the memory and synchronize it with the disk. */
    memset( addr, 'B', 5 );
    ret = msync( addr, file_size, MS_SYNC);
    if( ret == -1 ) {
        perror("msync");
        exit(0);
    }

    /* Close and reopen the file, and then read its contents. */
    close(fd);
    fd = open( filename, O_RDONLY);
    if( fd == -1 ) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    size_read = read( fd, buffer, sizeof( buffer ) );
    printf( "File content = %s\n", buffer );

    close(fd);
    return EXIT_SUCCESS;
}

```

Classification:

mmap() is [POSIX 1003.1 MFISHMITYM](#) (p. 46); *mmap64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mmap_device_io()

Gain access to a device's registers

Synopsis:

```
#include <stdint.h>
#include <sys/mman.h>

uintptr_t mmap_device_io( size_t len,
                          uint64_t io );
```

Arguments:

len

The number of bytes of device I/O memory that you want to access. It can't be 0.

io

The address of the area that you want to access.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mmap_device_io()` function maps *len* bytes of device I/O memory at *io* and makes it accessible via the [in*\(\)](#) (p. 1059) and [out*\(\)](#) (p. 1923) functions in `<hw/inout.h>`.



In order to map physical memory, your process must have the `PROCmgr_AID_MEM_PHYS` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

A handle to the device's I/O memory, or `MAP_DEVICE_FAILED` if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

Invalid *flags* type, or *len* is 0.

ENOMEM

The address range requested is outside of the allowed process address range, or there wasn't enough memory to satisfy the request.

ENXIO

The address from *io* for *len* bytes is invalid.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

You need I/O privileges to use the result of the *mmap_device_io()* function. The calling thread must:

- have the PROCmgr_AID_IO ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- call [ThreadCtl\(\)](#) (p. 3408) with the _NT0_TCTL_IO command to establish these privileges.

mmap_device_memory()

Map a device's physical memory into a process's address space

Synopsis:

```
#include <sys/mman.h>

void * mmap_device_memory( void * addr,
                           size_t len,
                           int prot,
                           int flags,
                           uint64_t physical );
```

Arguments:

addr

NULL, or a pointer to where you want to map the object in the calling process's address space.

len

The number of bytes you want to map into the caller's address space. It can't be 0.

prot

The access capabilities that you want to use for the memory region being mapped. You can use a combination of at least the following protection bits, as defined in `<sys/mman.h>`:

- `PROT_EXEC` — the region can be executed.

In order to successfully use this flag:

- Your process must have the `PROCmgr_AID_PROT_EXEC` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If your process has any privileged abilities enabled, then any memory-mapped files in the region must be from a trusted filesystem. For more information, see `pathtrust` in the *Utilities Reference*.

-
- `PROT_NOCACHE` — disable the caching of the region (e.g. to access dual-ported memory).



Read the architecture guide for your processor; you may need to add special instructions.

- `PROT_NONE` — the region can't be accessed.
- `PROT_READ` — the region can be read.
- `PROT_WRITE` — the region can be written.

flags

Specifies further information about handling the mapped region. You can use the following flag:

- `MAP_FIXED`

This function already uses `MAP_SHARED` (p. ?) ORed with `MAP_PHYS` (p. ?) (see `mmap()` for a description of these flags).

physical

The physical address of the memory to map into the caller's address space.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mmap_device_memory()` function maps *len* bytes of a device's *physical* memory address into the caller's address space at the location returned by `mmap_device_memory()`.

You should use this function instead of using `mmap()` (p. 1656) with the `MAP_PHYS` flag.



In order to map physical memory, your process must have the `PROCmgr_AID_MEM_PHYS` ability enabled. For more information, see `procmgr_ability()` (p. 2141).

Typically, you don't need to use *addr*; you can just pass `NULL` instead. If *addr* isn't `NULL` value, whether the object is mapped depends on whether or not you set `MAP_FIXED` in *flags*:

MAP_FIXED is set

The object is mapped to the address in *addr*, or the function fails. If this area is already mapped, the call changes the existing mapping of the area. A memory area being mapped with MAP_FIXED is first unmapped by the system using the same memory area. See [munmap\(\)](#) (p. 1832) for details.



- In order to use MAP_FIXED, your process must have the PROC_MGR_AID_MAP_FIXED ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
 - Use MAP_FIXED with caution. Not all memory models support it. In general, you should assume that you can MAP_FIXED only at an address (and size) that a call to *mmap()* without MAP_FIXED returned.
-

MAP_FIXED isn't set

The value of *addr* is taken as a hint as to where to map the object in the calling process's address space. The mapped area won't overlay any current mapped areas.

Returns:

The address of the mapped-in object, or MAP_FAILED if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

Invalid *flags* type, or *len* is 0.

ENOMEM

The address range requested is outside of the allowed process address range, or there wasn't enough memory to satisfy the request.

ENXIO

The address from *physical* for *len* bytes is invalid for the requested object, or MAP_FIXED was specified and *addr*, *len*, and *physical* are invalid for the requested object.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

Examples:

```
/* Map in the physical memory; 0xb8000 is text mode VGA video memory */
ptr = mmap_device_memory( 0, len, PROT_READ|PROT_WRITE|PROT_NOCACHE, 0, 0xb8000 );
if ( ptr == MAP_FAILED ) {
    perror( "mmap_device_memory for physical address 0xb8000 failed" );
    exit( EXIT_FAILURE );
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

modem_open()

Open a serial port

Synopsis:

```
#include <sys/modem.h>

int modem_open( char* device,
                speed_t baud );
```

Arguments:

device

The path name of the serial port that you want to open.

baud

Zero, or the baud rate that you want to use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `modem_open()` function opens a serial port identified by *device*. The device is set to raw mode by changing the control flags using `tcgetattr()` and `tcsetattr()` as follows:

```
termio.c_cflag = CS8|IHFLOW|OHFLOW|CREAD|HUPCL;
termio.c_iflag = BRKINT;
termio.c_lflag = IEXTEN;
termio.c_oflag = 0;
```

Any pending input or output characters are discarded.

If *baud* is nonzero, then the baud rate is changed to that value.

Returns:

An open file descriptor, or -1 on failure ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission is denied on a component of the path prefix, or the file doesn't exist.

EBADFSYS

While attempting to open the named file, either the file itself or a component of the path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You'll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY

The file named by *device* is a block special device that's already open for writing, or *device* names a file that's on a filesystem mounted on a block special device that's already open for writing, or *device* is in use.

EINTR

The open operation was interrupted by a signal.

EISDIR

The named *device* is a directory.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the *device* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENFILE

Too many files are currently open in the system.

ENOENT

The named *device* doesn't exist, or the *path* argument points to an empty string.

ENOSYS

The *modem_open()* function isn't implemented for the filesystem specified in *device*.

ENOTDIR

A component of the path prefix isn't a directory.

ENXIO

No process has the file open for reading.

Examples:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/modem.h>
#include <stdio.h>
#include <errno.h>

/*
curstate curflags newstate newflags newtimeout
newquiet retvalue pattern response
*/

struct modem_script table[] = {
    {1, 0, 1, 0, "ATZ\\r\\P0a"}, 2, 5, 0,
    {1, 0, 2, 0, "ATDT5910934"}, 30, 5, 0,
    {"*ok*", "ATDT5910934"},
    {2, MODEM_BAUD, 3, MODEM_LASTLINE, 10, 5, 0,
    {"*connect*", NULL},
    {3, 0, 4, 0, "guest"}, 8, 5, 0,
    {"*login:*", "guest"},
    {4, MODEM_NOECHO, 5, 0, 15, 5, 0,
    {"*password:*", "xxxx"},
    {5, 0, 0, 0, 0, 0, 0,
    {"*$*", NULL},
    {0, 0, 0, 0, 0, 0, 1,
    {"*no carrier*", NULL},
    {0, 0, 0, 0, 0, 0, 2,
    {"*no answer*", NULL},
    {0, 0, 0, 0, 0, 0, 3,
    {"*no dialtone*", NULL},
    {0, 0, 0, 0, 0, 0, 4,
    {"*busy*", NULL},
    { NULL }
};

void io(char* progress, char* in, char* out) {
    if(progress)
        printf("progress: %s\\n", progress);

    if(in)
        printf("input: %s\\n", in);

    if(out)
        printf("output: %s\\n", out);
}

int main(int argc, char* argv[]) {
    int fd, status;
    speed_t baud = -1;

    if((fd = modem_open(argv[1], 0)) == -1) {
        fprintf(stderr, "Unable to open %s: %s\\n",
            argv[1], strerror(errno));
        exit(1);
    }

    status = modem_script( fd, table, &baud,
        &io, NULL );
    printf("status=%d baud=%d\\n", status, baud);
}
```

```
    exit(status);  
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

modem_read()

Read bytes from a file descriptor

Synopsis:

```
#include <sys/modem.h>

int modem_read( int fd,
                char* buf,
                int bufsize,
                int quiet,
                int timeout,
                int flags,
                int (*cancel)(void) );
```

Arguments:

fd

The file descriptor for the device that you want to read from; see [modem_open\(\)](#) (p. 1672).

buf

A pointer to a buffer where the function can store the data.

bufsize

The size of the buffer, in bytes.

quiet

The maximum time to wait for more input after receiving at least one characters, in tenths of a second.

timeout

The maximum time to wait for any input, in tenths of a second.

flags

Flags that you can use to filter and map received characters; any combination of:

- **MODEM_ALLOWCASE** — preserve the case of incoming characters. Without this flag, all letters are mapped to lower case.
- **MODEM_ALLOWCTRL** — allow control characters. Without this flag, control characters are discarded.

- **MODEM_ALLOW8BIT** — preserve the top bit of incoming characters. Without this flag, the top bit is set to zero for all characters.
- **MODEM_LASTLINE** — discard all previously received characters when a newline is received followed by more characters. Without this flag, *buf* may contain multiple lines. If an automatic login script may be presented with an arbitrary text screen before the login prompt, you can use this flag to discard all but the login line, reducing the possibility of false matches.

cancel

NULL, or a callback that's called whenever the *quiet* time period expires while waiting for more input.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `modem_read()` function reads up to *bufsize* bytes from the device specified by the file descriptor, *fd*, and places them into the buffer pointed to by *buf*.

If no characters are received within the given *timeout*, `modem_read()` returns with -1.

When at least one character has been received, `modem_read()` returns if the flow of incoming characters stops for at least the *quiet* time period. The number of characters saved in *buf* is returned.

If you provide a *cancel* function, it's called once each *quiet* time period while waiting for input. If this function returns a nonzero value, `modem_read()` returns -1 immediately and sets *errno* to ETIMEDOUT. You can use the *cancel* function as a callback in a graphical dialer that needs to support a cancel button to stop a script (see [modem_script\(\)](#) (p. 1679)).

Returns:

Zero for success, or -1 on failure (*errno* (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set on this *fd*, and the process would have been blocked in trying to perform this operation.

EBADF

The argument *fd* is invalid, or the file isn't opened for reading.

EINTR

The *readcond()* call was interrupted by the process being signalled.

EIO

This process isn't currently able to read data from this *fd*.

ENOSYS

This function isn't supported for this *fd*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Read the <i>Caveats</i>

Caveats:

Depending on what you do in your *cancel* function, *modem_read()* may or not be signal handler or thread-safe.

modem_script()

Run a script on a device

Synopsis:

```
#include <sys/modem.h>

int modem_script( int fd,
                  struct modem_script* table,
                  speed_t* baud,
                  void (*io)(
                      char* progress,
                      char* in,
                      char* out ),
                  int (*cancel)(void) );
```

Arguments:

fd

The file descriptor for the device that you want to read from; see [modem_open\(\)](#) (p. 1672).

table

An array of `modem_script` structures that comprise a script of commands that you want to run on the device; see below.

baud

A pointer to a `speed_t` where the function can store the baud rate (if you script says to do so).

io

A function that's called to process each string that's emitted or received.

cancel

NULL, or a callback function that's called whenever the *newquiet* time period (specified in the script) expires while waiting for input.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `modem_script()` function runs the script *table* on the device associated with the file descriptor *fd*. The script implements a simple state machine that emits strings and waits for responses.

Each string that's emitted or received is passed to the function *io()* as follows:

Call	Description
<code>(*io)(str, 0, 0)</code>	Emitted <i>progress</i> string
<code>(*io)(0, str, 0)</code>	Received string
<code>(*io)(0, 0, str)</code>	Emitted <i>response</i> string

This lets an application set up a callback that can display the script's interaction in a status window.

If you provide a *cancel* function, it's called once each *newquiet* 1/10 of a second while waiting for input. If this function returns a nonzero value, the read returns immediately with -1 and *errno* is set to ETIMEDOUT. You can use the *cancel* function as a callback in a graphical dialer that needs to support a cancel button to stop a script.

The *table* is an array of `modem_script` structures that contain the following members:

char curstate

The current state. Execution always begins at [state 1](#) (p. 1682), which must be the first array element of *table*. Multiple elements may have the same current state, in which case any received input is matched against each *pattern* member for that state.

int curflags

The flags to use on a pattern match of a response:

- `MODEM_NOECHO` — don't echo the response through the *io()* callback.
- `MODEM_BAUD` — extract any number in the response and assign it to *baud*.

char newstate

When a pattern match occurs with *pattern*, this is the next state. A state transition causes *response* to be output and *newflags*, *newtimeout*, and

newquiet to be saved and associated with the new state. Changing to a new state of 0 causes *modem_script()* to return with the value in *retvalue*.

int newflags

Saved on a state transition and passed to *modem_read()* when waiting for a response in the new state. For information about these flags, see [modem_read\(\)](#) (p. 1676).

int newtimeout

Saved on a state transition and passed to *modem_read()* when waiting for a response in the new state. This timeout is described in [modem_read\(\)](#) (p. 1676).

int newquiet

Saved on a state transition and passed to *modem_read()* when waiting for a response in the new state. This quiet timeout is described in [modem_read\(\)](#) (p. 1676).

short retvalue

The return value when the script terminates with a pattern match, and the new state is 0.

char* pattern

A pattern to match against received characters. The pattern is matched using [fnmatch\(\)](#) (p. 677). Only patterns in the current state or the wildcard state of 0 are matched. On a match, the current state changes to *newstate*.

char* response

On a *pattern* match, this response is output to the device. If the *curflags* don't have `MODEM_NOECHO` set, the response is given to the callback function passed as the *io* parameter.

char* progress

On a *pattern* match, this progress string is passed to the callback function passed as the *io* parameter.

Here's an example that demonstrates the operation of the script:

```
/*
  curstate curflags newstate newflags newtimeout
  newquiet retvalue pattern response
*/
struct modem_script table[] ={
  {1, 0,          1, 0,          2,  5,  0,
   NULL,          "ATZ\\r\\P0a"},
  {1, 0,          2,  0,          30, 5,  0,
```

```

    "ok",          "ATDT5910934"},
    {2, MODEM_BAUD, 3, MODEM_LASTLINE, 10, 5, 0,
    "connect*",    NULL},
    {3, 0,          4, 0,                8, 5, 0,
    "login:",      "guest"},
    {4, MODEM_NOECHO, 5, 0,                15, 5, 0,
    "password:",   "xxxx"},
    {5, 0,          0, 0,                0, 0, 0,
    "$ ",          NULL},
    {0, 0,          0, 0,                0, 0, 1,
    "no carrier",  NULL},
    {0, 0,          0, 0,                0, 0, 2,
    "no answer",  NULL},
    {0, 0,          0, 0,                0, 0, 3,
    "no dialtone", NULL},
    {0, 0,          0, 0,                0, 0, 4,
    "busy",       NULL},
    { NULL }
};

```

When this script is passed to *modem_script()*, the current state is set to 1, and the output is ATZ (the *response* in the first array element).

While in any state, *modem_script()* waits for input, matching it against the current state or the wildcard state of 0.

State 1

Input	Action
ok	Go to state 2 (p. 1682) and emit ATDT1-591-0934. The flags to be used in the new state are set to 0, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 30 seconds.
no carrier	Go to state 0 (the termination newstate), return with the contents of <i>retvalue</i> (1).
no answer	Go to state 0 (the termination newstate), return with the contents of <i>retvalue</i> (2).
no dialtone	Go to state 0 (the termination newstate), return with the contents of <i>retvalue</i> (3).
busy	Go to state 0 (the termination newstate), return with the contents of <i>retvalue</i> (4).

State 2

Input	Action
connect	Go to state 3 (p. 1683) and don't emit anything to the device. The flags to be used in the new state are set to MODEM_LASTLINE, the quiet time in the new state is set to 5/10 of a second, and

Input	Action
	the timeout time in the new state is set to 10 seconds. Since the current flags are <code>MODEM_BAUD</code> , the baud rate is extracted from the connect message.
no carrier	Same as previous table
no answer	Same as previous table
no dialtone	Same as previous table
busy	Same as previous table

State 3

Input	Action
login	Go to state 4 (p. 1683) and emit <code>guest</code> . The flags to be used in the new state are set to 0, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 8 seconds.
no carrier	Same as previous table
no answer	Same as previous table
no dialtone	Same as previous table
busy	Same as previous table

State 4

Input	Action
password	Go to state 5 (p. 1684) and emit <code>xxxx</code> . The flags to be used in the new state are set to 0, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 15 seconds. Since the current flags are <code>MODEM_NOECHO</code> , the password response <code>xxxx</code> isn't sent to the <code>io()</code> callback.
no carrier	Same as previous table
no answer	Same as previous table
no dialtone	Same as previous table

Input	Action
busy	Same as previous table

State 5

Input	Action
*\$ *	Go to state 0 (the termination newstate), return with the contents of <i>retvalue</i> (0).
no carrier	Same as previous table
no answer	Same as previous table
no dialtone	Same as previous table
busy	Same as previous table

If you set the flag `MODEM_BAUD` for a state, then any number embedded in a matching response is extracted and assigned as a number to the *baud* parameter.

If you don't set the flag `MODEM_NOECHO` for a state, then all emitted strings are also given to the passed *io* function as `(*io)(0, 0, response)`.

Returns:

The *retvalue* member of a script entry that terminates the script. This will always be a positive number. If *modem_script* fails, it returns -1 and sets *errno* (p. 539).

Errors:**EAGAIN**

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor, *fildev*, isn't a valid file descriptor open for writing.

EINTR

The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EIO

A physical I/O error occurred. The precise meaning depends on the device.

EPIPE

An attempt was made to write to a pipe (or FIFO) that isn't open for reading by any process. A SIGPIPE signal is also sent to the process.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Read the <i>Caveats</i>

Caveats:

Depending on what you do in your *cancel* function, it might or might not be safe to call *modem_script()* from a signal handler or a multithreaded program.

modem_write()

Write a string to a device

Synopsis:

```
#include <sys/modem.h>

int modem_write( int fd,
                 char* str );
```

Arguments:

fd

The file descriptor for the device that you want to write to; see [modem_open\(\)](#) (p. 1672).

str

The string that you want to write.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `modem_write()` function writes the string `str` to the device specified by the file descriptor `fd`. Just before writing each character, all buffered input from the same device is flushed. After writing each character, an attempt to read an echo is made. The intent is to write a string without its appearing back in the input stream even if the device is echoing each character written.

If the `\` character appears in `str`, then the character following it is interpreted by `modem_write()`, and instead of both being written, they're treated as a special escape sequence that causes the following actions to be taken:

Escape	Description
<code>\r</code>	Output a carriage return.

Escape	Description
\n	Output a newline.
\xhh	Output the single character whose hex representation follows as <i>hh</i> .
\B	Send a 500 msec break on the line using <i>tcsendbreak()</i> .
\D	Drop the line for 1 second using <i>tcdropline()</i> .
\Phh	Pause for <i>hh</i> 1/10 of a second where <i>hh</i> is two hex characters.

Returns:

Zero on success, -1 on failure ([errno](#) (p. 539) is set).

Errors:**EAGAIN**

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor, *fildev*, isn't a valid file descriptor open for writing.

EINTR

The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EIO

A physical I/O error occurred. The precise meaning depends on the device.

EPIPE

An attempt was made to write to a pipe (or FIFO) that isn't open for reading by any process. A `SIGPIPE` signal is also sent to the process.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

modf(), modff()

Break a number into integral and fractional parts

Synopsis:

```
#include <math.h>

double modf( double value,
             double* iptr );

float modff( float value,
             float* iptr );
```

Arguments:

value

The value that you want to break into parts.

iptr

A pointer to a location where the function can store the integral part of the number.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The *modf()* and *modff()* functions break the given *value* into integral and fractional parts, each of which has the same sign as the argument. They store the integral part as a `double` in the object pointed to by *iptr*.

Returns:

The signed fractional part of *value*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    double integral_value, fractional_part;

    fractional_part = modf( 4.5, &integral_value );
    printf( "%f %f\n", fractional_part, integral_value );

    fractional_part = modf( -4.5, &integral_value );
    printf( "%f %f\n", fractional_part, integral_value );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0.500000 4.000000
-0.500000 -4.000000
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mount()

Mount a filesystem

Synopsis:

```
#include <sys/mount.h>

int mount( const char* spec,
           const char* dir,
           int flags,
           const char* type,
           const void* data,
           int datalen );
```

Arguments:

spec

A null-terminated string describing a special device (e.g. /dev/hd0t77), or NULL if there's no special device.

dir

A null-terminated string that names the directory that you want to mount (e.g. /mnt/home).

flags

Flags that are passed to the driver:

- `_MFLAG_OCB` — ignore the special device string, and contact all servers.
- `_MOUNT_READONLY` — mark the filesystem mountpoint as read-only.
- `_MOUNT_NOEXEC` — don't allow executables to load.
- `_MOUNT_NOSUID` — don't honor setuid bits on the filesystem.
- `_MOUNT_NOCREAT` — don't allow file creation on the filesystem.
- `_MOUNT_OFF32` — limit `off_t` to 32 bits.
- `_MOUNT_NOATIME` — disable logging of file access times.
- `_MOUNT_BEFORE` — call [resmgr_attach\(\)](#) (p. 2580) with `_RESMGR_FLAG_BEFORE`.
- `_MOUNT_AFTER` — call [resmgr_attach\(\)](#) with `_RESMGR_FLAG_AFTER`.
- `_MOUNT_OPAQUE` — call [resmgr_attach\(\)](#) with `_RESMGR_FLAG_OPAQUE`.
- `_MOUNT_UNMOUNT` — unmount this path.
- `_MOUNT_REMOUNT` — this path is already mounted; perform an update.
- `_MOUNT_FORCE` — force an unmount or a remount change.

- `_MOUNT_ENUMERATE` — autodetect on this device.

type

A null-terminated string with the filesystem type (e.g. `nfs`, `cifs`, `qnx4`, `ext2`, `network`).

data

A pointer to additional data to be sent to the manager. If *datalen* is `<0`, the data points to a null-terminated string.

datalen

The length of the data, in bytes, that's being sent to the server, or `<0` if the data is a null-terminated string.

Library:**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mount()* function sends a request to servers to mount the services provided by *spec* and *type* at *dir*.

Updating the mount resets the other mount flags to their default values. To determine which flags are currently set, use the `DCMD_ALL_GETMOUNTFLAGS devctl()` command, and then OR in `_MOUNT_REMOUNT`. For example:



```
int flags;
if(devctl(fd, DCMD_ALL_GETMOUNTFLAGS, &flags, sizeof flags, NULL) == EOK) {
    flags |= _MOUNT_REMOUNT;
    ...
}
```

If you set `_MFLAG_OCB` in the flags, then the special device string is ignored, and all servers are contacted. If you don't set this bit, and the special device *spec* exists, then only the server that created that device is contacted, and the full path to *spec* is provided.

If *datalen* is any value `<0`, and there's a data pointer, the function assumes that the data pointer is a pointer to a string.

Returns:

-1 on failure; no server supports the request ([errno](#) (p. 539) is set).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mount_parse_generic_args()

Strip off common mount arguments

Synopsis:

```
#include <sys/mount.h>

char * mount_parse_generic_args( char * options,
                                int * flags );
```

Arguments:

options

The string of options that you want to parse; see below.

flags

A pointer to a location where the function can store a set of bits corresponding to the options that it finds; see below.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *mount_parse_generic_args()* function parses the given *options*, removes any options that it recognizes, and sets or clears the appropriate bits in the location pointed to by *flags*.

This function lets you strip out common flags to help you parse mount arguments. This is useful when you want to create a custom mount utility.

Here's a list of the supported options that may be stripped:

Option:	Set/Clear this bit:	Description:
after	Set _MOUNT_AFTER	Call resmgr_attach() (p. 2580) with _RESMGR_FLAG_AFTER.
atime	Clear _MOUNT_NOATIME	Log file access times (default).

Option:	Set/Clear this bit:	Description:
before	Set _MOUNT_BEFORE	Call <i>resmgr_attach()</i> with _RESMGR_FLAG_BEFORE.
creat	Clear _MOUNT_NOCREAT	Allow file creation on the filesystem (default).
enumerate	Set _MOUNT_ENUMERATE	Auto-detect on this device.
exec	Clear _MOUNT_NOEXEC	Load executables (default).
force	Set _MOUNT_FORCE	Force an unmount or a remount change.
implied	Set _MOUNT_IMPLIED	The mountpoint is unspecified by the client.
noatime	Set _MOUNT_NOATIME	Disable logging of file access times.
nocreat	Set _MOUNT_NOCREAT	Don't allow file creation on the filesystem.
noexec	Set _MOUNT_NOEXEC	Don't allow executables to load.
nostat	Set _MFLAG_OCB	Don't attempt to <i>stat()</i> (p. 3164) the special device before mounting (i.e. -t).
nosuid	Set _MOUNT_NOSUID	Don't honor setuid bits on the filesystem.
opaque	Set _MOUNT_OPAQUE	Call <i>resmgr_attach()</i> with _RESMGR_FLAG_OPAQUE.
remount	Set _MOUNT_REMOUNT	This path is already mounted; perform an update.
ro	Set _MOUNT_READONLY	Mark the filesystem mountpoint as read-only.
rw	Clear _MOUNT_READONLY	Mark the filesystem mountpoint as read/write (default).
suid	Clear _MOUNT_NOSUID	Honor setuid bits on the filesystem (default).

Option:	Set/Clear this bit:	Description:
update	Set _MOUNT_REMOUNT	This path is already mounted; perform an update.

Returns:

A pointer to the modified *options* string, which contains any options that it didn't recognize, or NULL if the function processed all the options.

Examples:

```

while ((c = getopt(argv, argc, "o:")) {
    switch (c) {
        case 'o':
            if ((mysteryop = mount_parse_generic_args(optarg, &flags))) {
                /*
                 * You can do your own getsubopt type processing here.
                 * mysteryop doesn't contain the common options.
                 */
            }
            break;
    }
}

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mprobe()

Perform consistency check on memory

Synopsis:

```
#include <malloc.h>

enum mcheck_status mprobe(void * ptr);
```

Arguments:

ptr

A pointer to the start of the heap block.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *mprobe()* function attempts to perform consistency checks on the allocated block specified by *ptr*, which was previously returned by a call to [calloc\(\)](#) (p. 255), [malloc\(\)](#) (p. 1569) or [realloc\(\)](#) (p. 2527).

Consistency checks look for inconsistencies within the block header or in the block trailer byte. They may also detect block overruns.

The level of checking provided depends on which version of the allocator you've linked the application with:

- C library — minimal consistency checking.
- Nondebug version of the `malloc` library — a slightly greater level of consistency checking.
- Debug version of the `malloc` library — extensive consistency checking, with tuning available through the use of the [mallopt\(\)](#) (p. 1575) function.

Returns:

One of the values of the `mcheck_status` enumeration:

MCHECK_DISABLED

Consistency checking isn't currently enabled, or consistency information isn't available for this block.

MCHECK_OK

There are no inconsistencies in this block.

MCHECK_HEAD

The block header is corrupted.

MCHECK_TAIL

The block trailer byte is corrupted or there has been a block overrun.

MCHECK_FREE

The *ptr* argument doesn't point to an allocated heap block.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Calling *mprobe()* on a pointer already deallocated by a call to *free()* or *realloc()* could corrupt the memory allocator's data structures and result in undefined behavior.

mprotect()

Change memory protection

Synopsis:

```
#include <sys/mman.h>

int mprotect( void * addr,
              size_t len,
              int prot );
```

Arguments:

addr

The beginning of the range of addresses whose protection you want to change.

len

The length of the range of addresses, in bytes.

prot

The new access capabilities for the mapped memory region(s). You can combine the following bits, which are defined in `<sys/mman.h>`:

- **PROT_EXEC** — the region can be executed.

In order to successfully use this flag:

- Your process must have the **PROCmgr_AID_PROT_EXEC** ability enabled.
- If the calling process has any privileged abilities enabled, then any memory-mapped files in the region must be from a trusted filesystem. For more information about trusted filesystems, see **pathtrust** in the *Utilities Reference*.



For more information about abilities, see [*procmgr_ability\(\)*](#) (p. 2141).

- **PROT_NOCACHE** — disable caching of the region (for example, to access dual ported memory).
- **PROT_NONE** — the region can't be accessed.
- **PROT_READ** — the region can be read.
- **PROT_WRITE** — the region can be written.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `mprotect()` function changes the access protections on any mappings residing in the range starting at `addr`, and continuing for `len` bytes.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).



If `mprotect()` fails, the protections on some of the pages in the address range starting at `addr` and continuing for `len` bytes may have been changed.

Errors:**EACCES**

One of the following occurred:

- The memory object wasn't opened for read, regardless of the protection specified.
- The memory object wasn't opened for write, and you specified `PROT_WRITE` for a `MAP_SHARED` type mapping.
- You specified `PROT_EXEC` for a memory-mapped file mapping, the file doesn't have execute permission for the client process, and `procnto` was started with the `-mX` option.

EAGAIN

The `prot` argument specifies `PROT_WRITE` on a `MAP_PRIVATE` mapping, and there's insufficient memory resources to reserve for locking the private pages (if required).

ENOMEM

The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process, or specify one or more pages that are not mapped.

The *prot* argument specifies `PROT_WRITE` on a `MAP_PRIVATE` mapping, and locking the private pages (if required) would need more space than the system can supply to reserve for doing so.

ENOSYS

The function *mprotect()* isn't supported by this implementation.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or it attempted to set `PROT_EXEC` for a region of memory covered by an untrusted memory-mapped file.

Classification:

[POSIX 1003.1 MPR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mq_close()

Close a message queue

Synopsis:

```
#include <mqueue.h>

int mq_close( mqd_t mqdes );
```

Arguments:

mqdes

The message-queue descriptor, returned by [mq_open\(\)](#) (p. 1709), of the message queue that you want to close.

Library:

- For the traditional implementation, **libc**
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, **libmq**
Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_close()` function removes the association between `mqdes` and a message queue. If the current process attaches a notify to this queue for notification, the attachment is eliminated. If this queue is unlinked before the call to `mq_close()`, and this process is the last process to call `mq_close()` on the queue, then the queue is destroyed, along with its contents.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

In the traditional (`mqueue`) implementation, calling [close\(\)](#) (p. 336) with `mqdes` has the same effect as calling `mq_close()`.

Returns:

-1 if an error occurred ([errno](#) (p. 539) is set). Any other value indicates success.

Errors:**EBADF**

Invalid queue *mqdes*.

Classification:

[POSIX 1003.1 MSG](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_getattr()

Get a message queue's attributes

Synopsis:

```
#include <mqqueue.h>

int mq_getattr( mqd_t mqdes,
                struct mq_attr* mqstat );
```

Arguments:

mqdes

The message-queue descriptor, returned by [mq_open\(\)](#) (p. 1709), of the message queue that you want to get the attributes of.

mqstat

A pointer to a `mq_attr` structure where the function can store the attributes of the message queue. For more information, see below.

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_getattr()` function determines the current attributes of the queue referenced by `mqdes`. These attributes are stored in the location pointed to by `mqstat`.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

The fields of the `mq_attr` structure are as follows:

long *mq_flags*

The options set for this open message-queue description (i.e. these options are for the given *mqdes*, not the queue as a whole). This field may have been changed by call to [mq_setattr\(\)](#) (p. 1720) since you opened the queue.

- **O_NONBLOCK** — no call to [mq_receive\(\)](#) (p. 1714) or [mq_send\(\)](#) (p. 1717) will ever block on this queue. If the queue is in such a condition that the given operation can't be performed without blocking, then an error is returned, and *errno* is set to EAGAIN.

long mq_maxmsg

The maximum number of messages that can be stored on the queue. This value was set when the queue was created.

long mq_msgsize

The maximum size of each message on the given message queue. This value was also set when the queue was created.

long mq_curmsgs

The number of messages currently on the given queue.

long mq_sendwait

The number of threads currently waiting to send a message. This field was eliminated from the POSIX standard after draft 9, but has been kept as a QNX Neutrino extension. A nonzero value in this field implies that the queue is full.

long mq_rcvwait

The number of threads currently waiting to receive a message. Like *mq_sendwait*, this field has been kept as a QNX Neutrino extension. A nonzero value in this field implies that the queue is empty.



The alternate (mq) implementation of message queues doesn't see the non-POSIX *mq_sendwait* and *mq_rcvwait* fields.

Returns:

-1 if an error occurred (*errno* (p. 539) is set). Any other value indicates success.

Errors:

EBADF

Invalid message queue *mqdes*.

Classification:

POSIX 1003.1 MSG (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_notify()

Ask to be notified when there's a message in the queue

Synopsis:

```
#include <mqueue.h>

int mq_notify(
    mqd_t mqdes,
    const struct sigevent* notification );
```

Arguments:

mqdes

The message-queue descriptor, returned by [mq_open\(\)](#) (p. 1709), of the message queue that you want to get notification for.

notification

NULL, or a pointer to a [sigevent](#) (p. 2961) structure that describes how you want to be notified.

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

If *notification* isn't NULL, the *mq_notify()* function asks the server to notify the calling process when the queue makes the transition from empty to nonempty. The means by which the server is to notify the process is passed in the [sigevent](#) (p. 2961) structure pointed to by *notification*. Once the message queue server has notified the process of the transition, the notification is removed.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

We recommend that you use the following event types in this case:

- `SIGEV_SIGNAL`
- `SIGEV_SIGNAL_CODE`
- `SIGEV_SIGNAL_THREAD`
- `SIGEV_PULSE`

Under normal operation, only one process may register for notification at a time. If a process attempts to attach a notification, and another process is already attached, an error is returned and *errno* is set to `EBUSY`.

If a process has registered for notification, and another process is blocked on [mq_receive\(\)](#) (p. 1714), then the *mq_receive()* call is satisfied by any arriving message. The resulting behavior is as if the message queue remained empty.

If *notification* is `NULL` and the current process is currently registered for notification, then the existing registration is removed.

Returns:

-1 if an error occurred ([errno](#) (p. 539) is set). Any other value indicates success.

Errors:

EBADF

Invalid message queue *mqdes*.

EBUSY

A process has already registered for notification for the given queue.

Classification:

[POSIX 1003.1 MSG](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_open()

Open a message queue

Synopsis:

```
#include <mqueue.h>
#include <fcntl.h>

mqd_t mq_open( const char * name,
               int oflag,
               ... )
```

Arguments:

name

The name of the message queue that you want to open; see below.

oflag

You must specify one of **O_RDONLY** (receive-only), **O_WRONLY** (send-only) or **O_RDWR** (send-receive). In addition, you can OR in the following constants to produce the following effects:

- **O_CREAT** — if *name* doesn't exist, instruct the server to create a new message queue with the given name. If you specify this flag, *mq_open()* uses its *mode* and *mq_attr* arguments; see below.
- **O_EXCL** — if you set both **O_EXCL** and **O_CREAT**, and a message queue *name* exists, the call fails and *errno* is set to **EEXIST**. Otherwise, the queue is created normally. If you set **O_EXCL** without **O_CREAT**, it's ignored.
- **O_NONBLOCK** — under normal message queue operation, a call to *mq_send()* or *mq_receive()* could block if the message queue is full or empty. If you set this flag, these calls never block. If the queue isn't in a condition to perform the given call, *errno* is set to **EAGAIN** and the call returns an error.

If you set **O_CREAT** in the *oflag* argument, you must also pass these arguments to *mq_open()*:

mode

The file permissions for the new queue. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

If you set any bits other than file permission bits, they're ignored. Read and write permissions are analogous to receive and send permissions; execute permissions are ignored.

mq_attr

NULL, or a pointer to an `mq_attr` structure that contains the attributes that you want to use for the new queue. For more information, see [mq_getattr\(\)](#) (p. 1704).

If *mq_attr* is NULL, the following default attributes are used — depending on which implementation of message queues you're using — provided that you didn't override the defaults when you started the message-queue server:

Attribute	Traditional	Alternate
<i>mq_maxmsg</i>	1024	64
<i>mq_msgsize</i>	4096	256
<i>mq_flags</i>	0	0

If *mq_attr* isn't NULL, the new queue adopts the *mq_maxmsg* and *mq_msgsize* of `mq_attr`. The *mq_flags* flags field is ignored.

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_open()` function opens a message queue referred to by *name*, and returns a message queue descriptor by which the queue can be referenced in the future.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

The name is interpreted as follows:

- With the traditional (`mqueue`) implementation:

<i>name</i>	Pathname space entry
/entry	/dev/mqueue/entry
/entry/newentry	/entry/newentry
entry	cwd/entry
entry/newentry	cwd/entry/newentry

where *cwd* is the current working directory for the program at the point that it calls *mq_open()*.

- With the asynchronous (mq) implementation, message queues are always created under /dev/mq (or the path specified with the -N option to mq):
 - If the *name* argument starts with a slash, the queue is given that name.
 - If the *name* argument doesn't begin with a slash character, the queue is given that name, prepended with the current working directory.

In either case, slash characters other than the leading slash character aren't interpreted, and the specified name, including these slash characters, is used to identify the message queue. In other words, additional slashes don't create a directory structure under /dev/mq.

For example, if your current directory is /tmp:

<i>name</i>	Pathname space entry
/entry	/dev/mq/entry
entry	/dev/mq/tmp/entry



If you want to open a queue on another node, you have to use the traditional (mqueue) implementation and specify the name as */net/node/mqueue_location*.

If *name* doesn't exist, *mq_open()* examines the third and fourth parameters: a *mode_t* and a pointer to an *mq_attr* structure.

The only time that a call to *mq_open()* with *O_CREAT* set fails is if you open a message queue and later unlink it, but never close it. Like their file counterparts, an unlinked queue that hasn't yet been closed must continue to exist; an attempt to recreate such a message queue fails, and *errno* is set to *ENOENT*.



Message queues persist — like files — even after the processes that created them end. A message queue is destroyed when the last process connected to it unlinks from the queue by calling *mq_unlink()* (p. 1728).

Returns:

A valid message queue descriptor if the queue is successfully created, or -1 ([errno](#) (p. 539) is set).

Errors:**EACCES**

The message queue exists, and you don't have permission to open the queue under the given *oflag*, or the message queue doesn't exist, and you don't have permission to create one.

EEXIST

You specified the `O_CREAT` and `O_EXCL` flags in *oflag*, and the queue *name* exists.

EINTR

The operation was interrupted by a signal.

EINVAL

You specified the `O_CREAT` flag in *oflag*, and *mq_attr* wasn't `NULL`, but some values in the *mq_attr* structure were invalid.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Too many file descriptors are in use by the calling process.

ENAMETOOLONG

The length of *name* exceeds `PATH_MAX`.

ENFILE

Too many message queues are open in the system.

ENOENT

You didn't set the `O_CREAT` flag, and the queue *name* doesn't exist.

ENOSPC

The message queue server has run out of memory.

ENOSYS

The *mq_open()* function isn't implemented for the filesystem specified in *name*, or the message queue manager (`mq` or `mqueue`) isn't running.

Classification:

POSIX 1003.1 MSG (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_receive()

Receive a message from a queue

Synopsis:

```
#include <mqqueue.h>

ssize_t mq_receive( mqd_t mqdes,
                    char* msg_ptr,
                    size_t msg_len,
                    unsigned int* msg_prio );
```

Arguments:

mqdes

The message-queue descriptor, returned by [mq_open\(\)](#) (p. 1709), of the message queue that you want to get a message from.

msg_ptr

A pointer to a buffer where the function can store the message received.

msg_len

The message size of the given queue.

msg_prio

NULL, or a pointer to a location where the function can store the priority of the message received.

Library:

- For the traditional implementation, **libc**
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, **libmq**
Use the `-l mq` option to `qcc` to link against this library.

Description:

The *mq_receive()* function is used to receive the oldest of the highest priority messages in the queue specified by *mqdes*. The priority of the message received is put in the

location pointed to by *msg_prio*, the data itself in the location pointed to by *msg_ptr*, and the size received is returned.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for *mq* and *mqueue* in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

If you call *mq_receive()* with a *msg_len* that's less than the *mq_msgsize* of the specified queue, then *mq_receive()* returns an error and sets *errno* to *EMSGSIZE*.

If there are no messages on the queue specified, and *O_NONBLOCK* wasn't set (in the *oflag* argument to *mq_open()*), then the *mq_receive()* call blocks. If multiple *mq_receive()* calls are blocked on a single queue, then they're unblocked in FIFO order as messages arrive.

In the traditional (*mqueue*) implementation, calling *read()* (p. 2502) with *mqdes* is analogous to calling *mq_receive()* with a NULL *msg_prio*.

Returns:

The size of the message removed from the queue. If the call fails, -1 is returned as the size, no message is removed from the queue, and *errno* (p. 539) is set.

Errors:

EAGAIN

The *O_NONBLOCK* flag was set and there are no messages currently on the specified queue.

EBADF

The *mqdes* argument doesn't represent a valid queue open for reading.

EINTR

The operation was interrupted by a signal.

EINVAL

The *msg_ptr* argument isn't a valid pointer.

EMSGSIZE

The given *msg_len* is less than the *mq_msgsize* for the given queue or the given *msg_len* is too short for the message that would have been received.

Classification:

POSIX 1003.1 MSG (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_send()

Send a message to a queue

Synopsis:

```
#include <mqueue.h>

int mq_send( mqd_t mqdes,
             const char * msg_ptr,
             size_t msg_len,
             unsigned int msg_prio );
```

Arguments:

mqdes

The message-queue descriptor, returned by [mq_open\(\)](#) (p. 1709), of the message queue that you want to send a message to.

msg_ptr

A pointer to the message that you want to send.

msg_len

The size of the message.

msg_prio

The priority of the message, in the range from 0 to (MQ_PRIO_MAX-1).

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_send()` function puts a message of size `msg_len` and pointed to by `msg_ptr` into the queue indicated by `mqdes`. The new message has a priority of `msg_prio`.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

The queue is maintained in priority order, and in FIFO order within the same priority.

If the number of elements on the specified queue is equal to its `mq_maxmsg`, and `O_NONBLOCK` wasn't set (in the `oflag` argument to `mq_open()`), the call to `mq_send()` blocks. It becomes unblocked when there's room on the queue to send the given message. If more than one `mq_send()` is blocked on a given queue, and space becomes available in that queue to send, then the `mq_send()` with the highest priority message is unblocked.

In the traditional (`mqueue`) implementation, calling `write()` (p. 3737) with `mqdes` is analogous to calling `mq_send()` with a `msg_prio` of 0.

Returns:

-1 if an error occurred (`errno` (p. 539) is set). Any other value indicates success.

Errors:

EAGAIN

The `O_NONBLOCK` flag was set when opening the queue, and the specified queue is full.

EBADF

The `mqdes` argument doesn't represent a valid message queue descriptor, or `mqdes` wasn't opened for writing.

EINTR

The call was interrupted by a signal.

EINVAL

One of the following cases is true:

- `msg_len` was negative
- `msg_prio` was greater than `(MQ_PRIO_MAX-1)`
- `msg_prio` was less than 0

EMSGSIZE

The `msg_len` argument was greater than the `msgsize` associated with the specified queue.

Classification:

POSIX 1003.1 MSG (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_setattr()

Set a queue's attributes

Synopsis:

```
#include <mqueue.h>

int mq_setattr( mqd_t mqdes,
                const struct mq_attr* mqstat,
                struct mq_attr* omqstat );
```

Arguments:

mqdes

The message-queue descriptor, returned by [mq_open\(\)](#) (p. 1709), of the message queue that you want to set the attributes of.

mqstat

A pointer to a `mq_attr` structure that specifies the attributes that you want to use for the message queue. For more information about this structure, see [mq_getattr\(\)](#) (p. 1704); for information about which attributes you can set, see below.

omqstat

NULL, or a pointer to a `mq_attr` structure where the function can store the old attributes of the message queue.

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_setattr()` function sets the `mq_flags` field for the specified queue (passed as the `mq_flags` field in `mqstat`). If `omqstat` isn't NULL, then the old attribute structure is stored in the location that it points to.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

This function ignores the `mq_maxmsg`, `mq_msgsize`, and `mq_curmsgs` fields of `mqstat`. The `mq_flags` field is the bitwise OR of zero or more of the following constants:

O_NONBLOCK

No `mq_receive()` (p. 1714) or `mq_send()` (p. 1717) will ever block on this queue. If the queue is in such a condition that the given operation can't be performed without blocking, then an error is returned, and `errno` is set to `EAGAIN`.



The settings that you make for `mq_flags` apply only to the given message-queue description (i.e. locally), not to the queue itself.

Returns:

-1 if the function couldn't change the attributes (`errno` (p. 539) is set). Any other value indicates success.

Errors:

EBADF

Invalid message queue `mqdes`.

Classification:

[POSIX 1003.1 MSG](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mq_timedreceive(), mq_timedreceive_monotonic()

Receive a message from a message queue

Synopsis:

```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(
    mqd_t mqdes,
    char * msg_ptr,
    size_t msg_len,
    unsigned int * msg_prio,
    const struct timespec * abs_timeout );

ssize_t mq_timedreceive_monotonic(
    mqd_t mqdes,
    char * msg_ptr,
    size_t msg_len,
    unsigned int * msg_prio,
    const struct timespec * abs_timeout );
```

Arguments:

mqdes

The descriptor of the message queue you want to receive a message from, returned by [mq_open\(\)](#) (p. 1709).

msg_ptr

A pointer to a buffer where the function can store the message data.

msg_len

The size of the buffer, in bytes.

msg_prio

NULL, or a pointer to a location where the function can store the priority of the message that it removed from the queue.

abs_timeout

A pointer to a [timespec](#) (p. 3473) structure that specifies the *absolute* time (not the relative time to the current time) to wait before the function stops trying to receive messages.

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

The *mq_timedreceive()* function receives the oldest of the highest priority messages in the queue specified by *mqdes*. The *mq_timedreceive_monotonic()* function is a QNX Neutrino extension; it's similar to *mq_timedreceive()*, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the *Managing POSIX Message Queues* technote.

If you call *mq_timedreceive()* with a *msg_len* of anything other than the *mq_msgsize* of the specified queue, then *mq_timedreceive()* returns an error, and *errno* is set to `EINVAL`.

If there are no messages on the queue specified, and `O_NONBLOCK` wasn't set (in the *oflag* argument to *mq_open()*), then the *mq_timedreceive()* call blocks. If multiple *mq_timedreceive()* calls are blocked on a single queue, then they're unblocked in FIFO order as messages arrive.

In the traditional (`mqueue`) implementation, calling [read\(\)](#) (p. 2502) with *mqdes* is analogous to calling *mq_timedreceive()* with a `NULL msg_prio`.

Returns:

The size of the message removed from the queue, or -1 if an error occurred (no message is removed from the queue, and [errno](#) (p. 539) is set).

Errors:**EAGAIN**

The `O_NONBLOCK` flag was set and there are no messages currently on the specified queue.

EBADF

The *mqdes* argument doesn't represent a valid queue open for reading.

EINTR

The operation was interrupted by a signal.

EINVAL

One of the following:

- The *msg_ptr* argument isn't a valid pointer, or *msg_len* is less than 0, or *msg_len* is less than the message size specified in [mq_open\(\)](#) (p. 1709). The default message size is 4096 bytes.
- The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

EMSGSIZE

The given *msg_len* is shorter than the *mq_msgsize* for the given queue or the given *msg_len* is too short for the message that would have been received.

ETIMEDOUT

The timeout value was exceeded.

Examples:

Specify an absolute timeout of 1 second:

```
struct    timespec tm;
clock_gettime(CLOCK_REALTIME, &tm);
tm.tv_sec += 1;
if( 0 > mq_timedreceive( fd, buf, 4096, NULL, &tm ) ) {
    ...
}
```

Classification:

mq_timedreceive() is [POSIX 1003.1 MSG](#) (p. 46); *mq_timedreceive_monotonic()* is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mq_timedsend(), mq_timedsend_monotonic()

Send a message to a message queue

Synopsis:

```

#include <mqueue.h>
#include <time.h>

int mq_timedsend( mqd_t mqdes,
                  const char * msg_ptr,
                  size_t msg_len,
                  unsigned int msg_prio,
                  const struct timespec * abs_timeout );

int mq_timedsend_monotonic(
    mqd_t mqdes,
    const char * msg_ptr,
    size_t msg_len,
    unsigned int msg_prio,
    const struct timespec * abs_timeout );

```

Arguments:

mqdes

The descriptor of the message queue you want to put the message into, returned by [mq_open\(\)](#) (p. 1709).

msg_ptr

A pointer to the message data.

msg_len

The size of the buffer, in bytes.

msg_prio

The priority of the message, in the range from 0 to (MQ_PRIO_MAX-1).

abs_timeout

A pointer to a [timespec](#) (p. 3473) structure that specifies the *absolute* time (not the relative time to the current time) to wait before the function stops trying to receive messages.

Library:

- For the traditional implementation, `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

- For the alternate implementation using asynchronous messages, `libmq`

Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_timedsend()` function puts a message of size `msg_len` and pointed to by `msg_ptr` into the queue indicated by `mqdes`. The new message has a priority of `msg_prio`. The `mq_timedsend_monotonic()` function is a QNX Neutrino extension; it's similar to `mq_timedsend()`, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

The queue maintained is in priority order, and in FIFO order within the same priority.

If the number of elements on the specified queue is equal to its `mq_maxmsg`, and `O_NONBLOCK` wasn't set (in the `oflag` argument to `mq_open()`), the call to `mq_timedsend()` blocks. It becomes unblocked when there's room on the queue to send the given message. If more than one `mq_timedsend()` is blocked on a given queue, and space becomes available in that queue to send, then the `mq_timedsend()` with the highest priority message is unblocked.

In the traditional (`mqueue`) implementation, calling `write()` (p. 3737) with `mqdes` is analogous to calling `mq_timedsend()` with a `msg_prio` of 0.

Returns:

-1 if an error occurred (`errno` (p. 539) is set). Any other value indicates success.

Errors:

EAGAIN

The `O_NONBLOCK` flag was set when opening the queue, and the specified queue is full.

EBADF

The `mqdes` argument doesn't represent a valid message queue descriptor, or `mqdes` isn't opened for writing.

EINTR

The call was interrupted by a signal.

EINVAL

One of the following is true:

- The *msg_len* is negative.
- The *msg_prio* is greater than (MQ_PRIO_MAX-1).
- The *msg_prio* is less than 0.
- The MQ_PRIO_RESTRICT flag is set in the *mq_attr* member of *mqdes*, and *msg_prio* is greater than the priority of the calling process.
- The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

EMSGSIZE

The *msg_len* argument is greater than the *msgsize* associated with the specified queue.

ETIMEDOUT

The timeout value was exceeded.

Examples:

See the example for [mq_timedreceive\(\)](#) (p. 1724).

Classification:

mq_timedsend() is [POSIX 1003.1 MSG](#) (p. 46); *mq_timedsend_monotonic()* is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

mq_unlink()

Remove a queue

Synopsis:

```
#include <mqueue.h>

int mq_unlink( const char* name );
```

Arguments:

name

The name of the message queue that you want to unlink.

Library:

- For the traditional implementation, `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
Use the `-l mq` option to `qcc` to link against this library.

Description:

The *mq_unlink()* function removes the queue with the given *name*.



QNX Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entries for `mq` and `mqueue` in the *Utilities Reference*, as well as the Managing POSIX Message Queues technote.

If some process has the queue open when the call to *mq_unlink()* is made, then the actual deletion of the queue is postponed until it has been closed. If a queue exists in the netherworld between unlinking and the actual removal of the queue, then *all* calls to open a queue with the given name fail (even if `O_CREAT` is present in *oflag*). Once the queue is deleted, all elements currently on it are freed. Due to the lazy deletion of queues, it's impossible for any process to be blocked on the message queue when it's deleted.

Calling [unlink\(\)](#) (p. 3579) with a name that resolves to the message queue server's namespace (e.g. /dev/mqueue/my_queue) is analogous to calling `mq_unlink()` with *name* set to the last elements of the pathname (e.g. my_queue).

Returns:

-1 if the queue wasn't successfully unlinked ([errno](#) (p. 539) is set). Any other value indicates that the queue was successfully unlinked.

Errors:

EACCES

You don't have permission to unlink the specified queue.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *name* exceeds PATH_MAX.

ENOENT

The queue *name* doesn't exist.

ENOSYS

The `mq_unlink()` function isn't implemented for the filesystem specified in *name*, or the message queue manager (mq or mqueue) isn't running.

Classification:

[POSIX 1003.1 MSG](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

mrand48()

Generate a pseudo-random signed long integer

Synopsis:

```
#include <stdlib.h>

long mrand48( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *mrand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a signed long integer uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Call one of [lcong48\(\)](#) (p. 1505), [seed48\(\)](#) (p. 2787), or [srand48\(\)](#) (p. 3157) to initialize the random-number generator before calling [drand48\(\)](#) (p. 468), [lrand48\(\)](#) (p. 1551), or *mrand48()*.

The [jrand48\(\)](#) (p. 1492) function is a thread-safe version of *mrand48()*.

Returns:

A pseudo-random long integer.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

_msg_info

Information about a message

Synopsis:

```
#include <sys/neutrino.h>

struct _msg_info {          /* _msg_info    _server_info */
    uint32_t nd;            /* client      server */
    uint32_t srcnd;         /* server      n/a */
    pid_t pid;              /* client      server */
    int32_t tid;            /* thread      n/a */
    int32_t chid;           /* server      server */
    int32_t scoid;          /* server      server */
    int32_t coid;           /* client      client */
    int32_t msglen;         /* msg         n/a */
    int32_t srcmsglen;      /* thread      n/a */
    int32_t dstmsglen;      /* thread      n/a */
    int16_t priority;       /* thread      n/a */
    int16_t flags;          /* n/a         client */
    uint32_t reserved;
};
```

Description:

The `_msg_info` structure contains information about a message. The members include:

nd

The node descriptor of the client machine as viewed by the server. See “[Node descriptors](#) (p. 1733),” below.

srcnd

The node descriptor of the server, as viewed by the client.

pid

The process ID of the sending thread.

tid

The thread ID of the sending thread.

chid

The channel ID that the message was received on.

scoid

The server connection ID. This is a unique server-side client process identifier, corresponding to a particular process ID on a particular node.

coid

The client connection ID.

msglen

The number of bytes received.

srcmsglen

The length of the source message, in bytes, as sent by *MsgSend*()*. This may be greater than the value in *msglen*. This member is valid only if you set `_NTO_CHF_SENDER_LEN` in the *flags* argument to *ChannelCreate()* (p. 278) for the channel that received the message.

dstmsglen

The length of the client's reply buffer, in bytes, as passed to *MsgSend*()*. This member is valid only if you set `_NTO_CHF_REPLY_LEN` in the *flags* argument to *ChannelCreate()* (p. 278) for the channel that received the message.

priority

The priority of the sending thread.

flags

Flags that provide information about the sender and message:

- `_NTO_MI_ENDIAN_BIG` — the sending machine is big endian.
- `_NTO_MI_ENDIAN_DIFF` — the sending machine has a different endian-ness from the receiving machine.
- `_NTO_MI_CONSTRAINED` (QNX Neutrino 6.6 or later) — the message was received from a resource-constrained client. A thread that allocates resource on behalf of a constrained client should temporarily constrain itself too; see Resource constraint thresholds in the Processes chapter of the QNX Neutrino *Programmer's Guide*.
- `_NTO_MI_NET_CRED_DIRTY` — used internally.
- `_NTO_MI_UNBLOCK_REQ` — the client has an unblock pending (i.e. a timeout on the send occurred or a signal was delivered and `_NTO_CHF_UNBLOCK` is set on the channel).



The *msglen* and *srcmsglen* members are valid only until the next call to *MsgRead*()* or *MsgWrite*()*.

If *msglen* is less than *srcmsglen* and is also less than the receive buffer size, the message is a network transaction that requires more reading of data with *MsgRead*()*.

Node descriptors

The *nd* (node descriptor) is a temporary numeric description of a remote node; ND_LOCAL_NODE (or 0) is the descriptor for the local node. For more information, see the Qnet Networking chapter of the *System Architecture* guide.

To:	Use this function:
Compare two <i>nd</i> objects	ND_NODE_CMP() (p. 1874)
Convert a <i>nd</i> to text	netmgr_ndtostr() (p. 1877)
Convert text to a <i>nd</i>	netmgr_strtond() (p. 1883)

Classification:

[QNX Neutrino](#) (p. 46)

MsgCurrent(), MsgCurrent_r()

Resume processing of a message

Synopsis:

```
int MsgCurrent( int rcvid );  
int MsgCurrent_r( int rcvid );
```

Arguments:

rcvid

The receive ID that *MsgReceive*()* returned when you received the message.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgCurrent()* kernel call announces to the kernel that you are resuming the processing of a previously received message. The kernel checks that the thread identified by *rcvid* is still valid and is reply-blocked on the connection associated with that *rcvid*.

If the channel doesn't have `_NTO_CHF_FIXED_PRIORITY` set in its flags, then the appropriate priority adjustment will be made to the calling thread's priority.

Blocking states

None. If priority inheritance causes the priority of the calling thread to drop, other threads may run.

Returns:

The only difference between the *MsgCurrent()* and *MsgCurrent_r()* functions is the way they indicate errors:

MsgCurrent()

If an error occurs, -1 is returned and *errno* (p. 539) is set; otherwise, 0 is returned.

MsgCurrent_r()

The number of bytes read. This function does **NOT** set *errno*. If an error occurs, the value from the Errors section is returned, otherwise 0 is returned.

Errors:**ESRCH**

The thread associated with the *rcvid* is not in either *STATE_REPLY* or *STATE_NET_REPLY*, or it is not blocked on the connection associated with the *rcvid*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgDeliverEvent(), MsgDeliverEvent_r()

Deliver an event through a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgDeliverEvent( int rcvid,
                    const struct sigevent* event );

int MsgDeliverEvent_r(
                    int rcvid,
                    const struct sigevent* event );
```

Arguments:

rcvid

The value returned to the server when it receives a message from a client using *MsgReceive*()*, or 0 to deliver the event to the active thread (see below).

event

A pointer to a [sigevent](#) (p. 2961) structure that contains the event you want to send. These events are defined in `<sys/siginfo.h>`. The type of event is placed in *event.sigev_notify*.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgDeliverEvent()* and *MsgDeliverEvent_r()* kernel calls deliver an *event* from a server to a client through a channel connection. They're typically used to perform async IO and async event notification to clients that don't want to block on a server.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1739) section for details.

Although the server can explicitly send any event it desires, it's more typical for the server to receive a `struct sigevent` in a message from the client that already contains this data. The message also contains information for the server indicating

the conditions on when to notify the client with the event. The server then saves the *rcvid* from *MsgReceive*()* and the event from the message without needing to examine the event in any way. When the trigger conditions are met in the server, such as data becoming available, the server calls *MsgDeliverEvent()* with the saved *rcvid* and *event*.

You can use the SIGEV_SIGNAL set of notifications to create an asynchronous design in which the client is interrupted when the event occurs. The client can make this synchronous by using the [SignalWaitinfo\(\)](#) (p. 2997) kernel call to wait for the signal. Where possible, you should use an event-driven synchronous design that's based on SIGEV_PULSE. In this case, the client sends messages to servers, and requests event notification via a pulse.

You're not likely to use the event types SIGEV_UNBLOCK and SIGEV_INTR with this call.

You should use *MsgDeliverEvent()* when two processes need to communicate with each other without the possibility of deadlock. The blocking nature of *MsgSend*()* introduces a hierarchy of processes in which “sends” flow one way and “replies” the other way.

In the following diagram, processes at the A level can send to processes at the B or C level. Processes at the B level can send to the C level but they should never send to the A level. Likewise, processes at the C level can never send to those at the A or B level. To A, B and C are servers. To B, A is a client and C is a server.

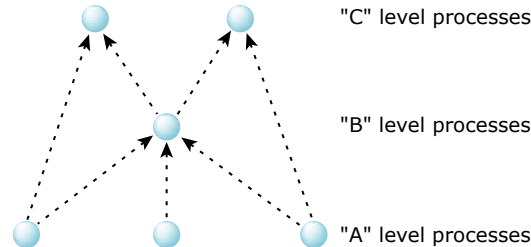


Figure 2: A hierarchy of processes.

These hierarchies are simple to establish and ensure a clean deadlock-free design. If these rules are broken then deadlock can occur as shown below:

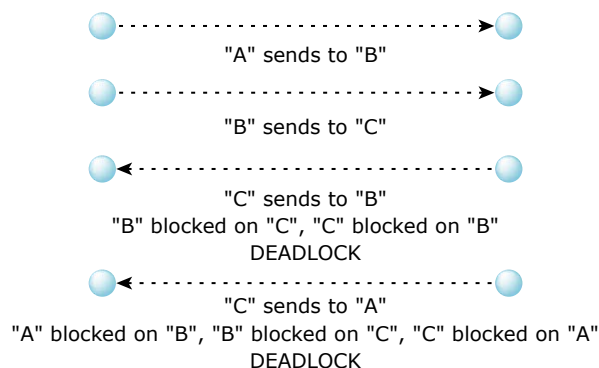


Figure 3: A deadlock when sending messages improperly among processes.

There are common situations which require communication to flow backwards through the hierarchy. For example, A sends to B requesting notification when data is available. B immediately replies to A. At some point in the future, B will have the data A requested and will inform A. B can't send a message to A because this might result in deadlock if A decided to send to B at the same time.

The solution is to have B use a nonblocking *MsgDeliverEvent()* to inform A. A receives this pulse and sends a message to B requesting the data. B then replies with the data. This is the basis for asynchronous IO. Clients send to servers and where necessary, servers use pulses to request clients to resend to them as needed. This is illustrated below:

Message	Use
A sends to B	Async IO request
B replies to A	Request acknowledged
B sends pulse to A	Requested data available
A sends to B	Request for the data
B replies to A	Reply with data



In client/server designs, you typically use *MsgDeliverEvent()* in the server, and *MsgSendPulse()* (p. 1789) in the client.

Sometimes it might be convenient for a library that doesn't have the *rcvid* to deliver the event instead of handing it off to the server. The library could interpret the *sigevent* itself, but this duplicates what the kernel already knows how to do. Instead, the library can call *MsgDeliverEvent()* with a *rcvid* of 0. The kernel then creates an artificial *rcvid* that references the calling thread — as if a server had received the message from it via *MsgReceive()* — and then the rest of the *MsgDeliverEvent()* proceeds normally.

If the client program replaces its process image (e.g., by calling *exec*()* (p. 487)) with a *setuid* or *setgid* binary, the kernel sets the *_NTO_COF_INSECURE* flag on the channel. Until the process that owns the channel uses *ConnectFlags()* (p. 369) to clear this flag, calls to *MsgDeliverEvent()* fail with an error of *EACCES*.

Blocking states

None. In the network case, lower priority threads may run.

Native networking

When you use *MsgDeliverEvent()* to communicate across a network, the return code isn't "reliable". In the local case, *MsgDeliverEvent()* always returns a correct success or failure value. But since *MsgDeliverEvent()* *must be nonblocking*, in the networked case, the return value isn't guaranteed to reflect the actual result on the client's node.

This is because *MsgDeliverEvent()* would have to block waiting for the communications between the two `lsm-qnet.so` objects.

Generally, this isn't a problem, because *MsgDeliverEvent()* is for the benefit of the client anyway — if the client no longer exists, then the client obviously doesn't care that it didn't get the event. The server usually delivers the event and then goes about its business, regardless of the success or failure of the event delivery.

Returns:

The only difference between these functions is the way they indicate errors:

MsgDeliverEvent()

If an error occurs, -1 is returned and *errno* (p. 539) is set. Any other value returned indicates success.

MsgDeliverEvent_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EACCES

The client program replaced its process image (e.g., by calling *exec*()* (p. 487)) with a `setuid` or `setgid` binary, so the kernel has set the `_NTD_COF_INSECURE` flag on the channel. The process that owns the channel needs to call *ConnectFlags()* (p. 369) to clear this flag.

EAGAIN

The kernel has insufficient resources to enqueue the event.

EBADF

The thread indicated by *rcvid* had its connection detached.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The given event isn't valid. For example, the type of notification might be invalid, or the signal for a `SIGEV_SIGNAL` event might be out of range.

ESRCH

The thread indicated by *rcvid* doesn't exist.

ESRVRFAULT

A fault occurred in the server's address space when the kernel tried to write the pulse message to the server's receive message buffer (SIGEV_PULSE only).

Examples:

The following example demonstrates how a client can request a server to notify it with a pulse at a later time (in this case, after the server has slept for two seconds). The server side notifies the client using *MsgDeliverEvent()*.

Here's the header file that's used by `client.c` and `server.c`:

```
struct my_msg
{
    short type;
    struct sigevent event;
};

#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL+5
#define MSG_GIVE_PULSE _IO_MAX+4
#define MY_SERV "my_server_name"
```

Here's the client side that fills in a `struct sigevent` and then receives a pulse:

```
/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>

#include "my_hdr.h"

int main( int argc, char **argv)
{
    int chid, coid, srv_coid, rcvid;
    struct my_msg msg;
    struct _pulse pulse;

    /* we need a channel to receive the pulse notification on */
    chid = ChannelCreate( 0 );

    /* and we need a connection to that channel for the pulse to be
       delivered on */
    coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );

    /* fill in the event structure for a pulse */
    SIGEV_PULSE_INIT( &msg.event, coid, SIGEV_PULSE_PRIO_INHERIT,
                     MY_PULSE_CODE, 0 );
    msg.type = MSG_GIVE_PULSE;

    /* find the server */
    if ( (srv_coid = name_open( MY_SERV, 0 )) == -1)
    {
        printf("failed to find server, errno %d\n", errno );
        exit(1);
    }

    /* give the pulse event we initialized above to the server for
       later delivery */
    MsgSend( srv_coid, &msg, sizeof(msg), NULL, 0 );

    /* wait for the pulse from the server */
    rcvid = MsgReceivePulse( chid, &pulse, sizeof( pulse ), NULL );
    printf("got pulse with code %d, waiting for %d\n", pulse.code,
          MY_PULSE_CODE );
```



```
    return 0;
}
```

Here's the server side that delivers the pulse defined by the struct sigevent:

```
/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

#include "my_hdr.h"

int main(int argc, char **argv) {
    int rcvid;
    union {
        struct my_msg mine;
        struct _pulse pulse;
    } msg;
    name_attach_t *attach;

    /* Attach the name the client will use to find us. */
    /* Our channel will be in the attach structure */
    if ((attach = name_attach(NULL, MY_SERV, 0)) == NULL) {
        printf("server:failed to attach name, errno %d\n", errno);
        exit(1);
    }

    /* Wait for the message from the client. */
    for(;;) {
        rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);
        switch(msg.mine.type) {
            case _PULSE_CODE_DISCONNECT:
                ConnectDetach(msg.pulse.scoid);
                break;
            case _IO_CONNECT:
                MsgReply(rcvid, 0, NULL, 0);
                break;
            case MSG_GIVE_PULSE:
                /* Wait until it is time to notify the client */
                sleep(2);

                /*
                 * Deliver notification to client that client
                 * requested
                 */
                MsgDeliverEvent(rcvid, &msg.mine.event);
                printf("server:delivered event\n");
                return 0;
            default:
                printf("server: unexpected message %d\n", msg.mine.type);
                return 0;
        }
    }
    return 0;
}
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

In the case of a pulse event, if the server faults on delivery, the pulse is either lost or an error is returned.

MsgError(), MsgError_r()

Unblock a client and set its errno

Synopsis:

```
#include <sys/neutrino.h>

int MsgError( int ravid,
              int error );

int MsgError_r( int ravid,
                int error );
```

Arguments:

ravid

The receive ID that *MsgReceive*()* returned.

error

The error code that you want to set for the client.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgError()* and *MsgError_r()* kernel calls unblock the client's *MsgSend*()* call and set the client's [errno](#) (p. 539) to *error*. No data is transferred.

If *error* is EOK, the *MsgSend*()* call returns EOK; if *error* is any other value, the *MsgSend*()* call returns -1.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1744) section for details.



An error number of ERESTART causes the sender to immediately call *MsgSend*()* again. Since send and reply buffers passed to [MsgSend\(\)](#) (p. 1781) may overlap, you shouldn't use ERESTART after a call to [MsgWrite\(\)](#) (p. 1819).

Blocking states

None. In the network case, lower priority threads may run.

Native networking

MsgError() has increased latency when you use it to communicate across a network — the server is now writing the error code to its local `lsm-qnet.so`, which may need to communicate with the client's `lsm-qnet.so` to actually transfer the error code.

Returns:

The only difference between these functions is the way they indicate errors:

MsgError()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

MsgError_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function returns one of the values listed in the Errors section.

Errors:

ESRCH

The thread indicated by *rcvid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgInfo(), MsgInfo_r()

Get additional information about a message

Synopsis:

```
#include <sys/neutrino.h>

int MsgInfo( int ravid,
             struct _msg_info* info );

int MsgInfo_r( int ravid,
               struct _msg_info* info );
```

Arguments:

ravid

The return value from *MsgReceive*()*.

info

A pointer to a [_msg_info](#) (p. 1731) structure where the function can store information about the message.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgInfo()* and *MsgInfo_r()* kernel calls get additional information about a received message and store it in the specified `_msg_info` structure.

These functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 1746) section for details.



The `info->msglen` and `info->srcmsglen` members are valid only until the next call to *MsgRead*()* or *MsgWrite*()*.

Blocking states

This call doesn't block.

Returns:

The only difference between these functions is the way they indicate errors:

MsgInfo()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

MsgInfo_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

ESRCH

The thread indicated by *rcvid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgKeyData(), MsgKeyData_r()

Pass data through a common client

Synopsis:

```
#include <sys/neutrino.h>

int MsgKeyData( int rcvid,
                int op,
                uint32_t key,
                uint32_t * key2,
                const iov_t * msg,
                int parts );

int MsgKeyData_r( int rcvid,
                  int op,
                  uint32_t key,
                  uint32_t * key2,
                  const iov_t * msg,
                  int parts );
```

Arguments:***rcvid***

The return value from *MsgReceive*()*.

op

The operation to perform; one of:

- `_NTO_KEYDATA_CALCULATE` — calculate a new key.
- `_NTO_KEYDATA_VERIFY` — verify the key.

key

A private value for key. The value that you use depends on your system's security requirements. If security is important, you should use a value read from `/dev/random` (see the entry for `random` in the *Utilities Reference*); otherwise you might choose to use a statistically random number returned by [rand\(\)](#) (p. 2484).

key2

A pointer to a key. What the function stores in this location depends on the *op* argument:

- `_NTO_KEYDATA_CALCULATE` — the new key.

- `_NTO_KEYDATA_VERIFY` — zero if no tampering has occurred.

msg

A pointer to a portion of the reply data to be keyed.

parts

The number of parts in *msg*.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgKeyData()* and *MsgKeyData_r()* kernel calls allow two privileged processes to pass data through a common client while verifying that the client hasn't modified the data. This is best explained by an example.



In order to successfully call these functions, your process must have the `PROCmgr_AID_KEYDATA` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1750) section for details.

A program calls [open\(\)](#) (p. 1905) with a filename. The *open()* function sends a message to the Process Manager, which is responsible for pathname management.

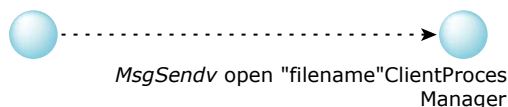


Figure 4: *MsgSendv()*, client to process manager.

The Process Manager resolves the pathname, resulting in a fully qualified network path and the process ID to send the *open()* request to. This information is replied to the client.



Figure 5: *MsgReplyv()*, process manager to client.

The client now sends this message to *pid* with the fully qualified pathname.



Figure 6: *MsgSendv()*, client to filesystem manager

Note that the client can change the pathname before it sends it to the Filesystem Manager. In fact, it could skip the call to the Process Manager and manufacture any pathname it desired. The Filesystem Manager always performs permission checking. Therefore, changing or manufacturing pathnames isn't normally something to be concerned about, except in one case: *chroot()* (p. 295) lets you specify a prefix that must be applied to all pathnames.

In the above example, the client may have had a *chroot()* of `/net/node2/home/dan`. This should limit the process from accessing files outside of `/net/node2/home/dan`. For example:

User path	Mapped to <i>chroot()</i> path
<code>/bin/ls</code>	<code>/net/node2/home/dan/bin/ls</code>
<code>/</code>	<code>/net/node2/home/dan</code>

The process has had its root set to a subdirectory, limiting the files it can access. For this to work, it's necessary to prevent the client from changing or manufacturing its own pathnames.



In the QNX Neutrino RTOS, only the Process Manager handles a user *chroot()* (p. 295). Unlike a monolithic kernel where the filesystem shares the same address space as the kernel and the *chroot()* information, QNX Neutrino I/O managers reside in separate address spaces and might not even reside on the same machine.

The solution to this problem is the *MsgKeyData()* call. When the Process Manager receives the *open()* (p. 1905) message, it generates the reply data. Before replying, it calls *MsgKeyData()*, with these arguments:

rcvid

The return value from *MsgReceive*()*.

op

`_NTO_KEYDATA_CALCULATE`

key

A private value for the key.

key2

A pointer to a new key that should be returned to the client in a unkeyed area of the message.

msg

A pointer to a portion of the reply data to be keyed.

parts

The number of parts in *msg*.

The client now sends the message to the File Manager. On receipt of the message, the File Manager calls *MsgKeyData()* with the same arguments as above, except for:

op

_NTO_KEYDATA_VERIFY

key

The key that's provided in the message.

MsgKeyData() sets the key pointed to by *key2* to zero if no tampering has occurred.

Note that there are actually two keys involved. A public key that's returned to the client and a private key that the Process Manager generated. The algorithm uses both keys and the data for verification.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

MsgKeyData()

If an error occurs, -1 is returned and *errno* (p. 539) is set. Any other value returned indicates success.

MsgKeyData_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:**EPERM**

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The thread indicated by *rcvid* doesn't exist.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

Examples:

```

/*
 * This program demonstrates the use of MsgKeyData() as a way
 * of a client handing off data from a source server to a
 * destination server such that if the client tampers with
 * the data, the destination server will know about it.
 */

#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/neutrino.h>

typedef struct {
    int    public_key;
    char    text[10];
} IPC_t;

int chid_src, chid_dst;

void* server_src_thread(void* parm);
void* server_dst_thread(void* parm);

main()
{
    pthread_t    tid[2];
    IPC_t        msg;
    int          coid;
    int          status;

    pthread_create(&tid[0], NULL, server_src_thread, NULL);
    pthread_create(&tid[1], NULL, server_dst_thread, NULL);
    sleep(3);
    /* give time for channels to be created, sloppy but simple */

    /*
     * Send to server_src_thread for some data.
     * The data will include some text and a public
     * key for that text.
     */

    coid = ConnectAttach(0, 0, chid_src, 0, 0);
    MsgSend(coid, NULL, 0, &msg, sizeof(msg));
    ConnectDetach(coid);

    /*
     * Now send to server_dst_thread with the reply from
     * server_src_thread. We didn't modify the 'text' so it
     * should reply success. Note that we're including the
     * public key.
     */

    coid = ConnectAttach(0, 0, chid_dst, 0, 0);
    status = MsgSend(coid, &msg, sizeof(msg), &msg, sizeof(msg));
    printf("Sent unmodified text to server_dst_thread.\n");
    printf("Replied with %s\n", status == EOK ? "EOK" : "EINVAL" );
}

```

```

/*
 * Now tamper with the original 'text' (which we aren't
 * supposed to do) and send to server_dst_thread again
 * but with the modified 'text' and the public key.
 * Since we tampered with the 'text', server_dst_thread
 * should reply failure.
 */

strcpy(msg.text, "NEWDATA");
status = MsgSend(coid, &msg, sizeof(msg), &msg, sizeof(msg));
printf("Sent modified text to server_dst_thread.
      Replied with %s\n", status == EOK ? "EOK" : "EINVAL" );

return 0;
}

void* server_src_thread(void* parm)
{
    int          rcvid;
    int          private_key; /* the kernel keeps this */
    iov_t        keyed_area_iov;
    IPC_t        msg;
    struct timespec t;

    chid_src = ChannelCreate(0);
    while (1) {
        rcvid = MsgReceive(chid_src, &msg, sizeof(msg), NULL);

        /*
         * Give MsgKeyData() the private key and it will
         * calculate a public key for the 'text' member of
         * the message. The kernel will keep the private key
         * and we reply with the public key.
         * Note that we use the number of nanoseconds since the
         * last second as a way of getting a 32-bit pseudo
         * random number for the private key.
         */

        clock_gettime(CLOCK_REALTIME, &t);
        private_key = t.tv_nsec; /* nanoseconds since last second */
        strcpy(msg.text, "OKDATA");
        SETIOV(&keyed_area_iov, &msg.text, sizeof(msg.text));
        MsgKeyData(rcvid, _NTO_KEYDATA_CALCULATE, private_key,
                  &msg.public_key, &keyed_area_iov, 1);

        MsgReply(rcvid, 0, &msg, sizeof(msg));
    }
    return NULL;
}

void* server_dst_thread(void* parm)
{
    int          rcvid, tampered, status;
    iov_t        keyed_area_iov;
    IPC_t        msg;

    chid_dst = ChannelCreate(0);
    while (1) {
        rcvid = MsgReceive(chid_dst, &msg, sizeof(msg), NULL);

        /*
         * Use the public key to see if the data
         * has been tampered with.
         */

        SETIOV(&keyed_area_iov, &msg.text, sizeof(msg.text));
        MsgKeyData(rcvid, _NTO_KEYDATA_VERIFY, msg.public_key,
                  &tampered, &keyed_area_iov, 1);

        if (tampered)
            status = EINVAL; /* reply: 'text' was modified */
        else
            status = EOK; /* reply: 'text' was okay */
        MsgReply(rcvid, status, &msg, sizeof(msg));
    }
    return NULL;
}

```

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgRead(), MsgRead_r()

Read data from a message

Synopsis:

```
#include <sys/neutrino.h>

int MsgRead( int rcvid,
             void* msg,
             int bytes,
             int offset );

int MsgRead_r( int rcvid,
               void* msg,
               int bytes,
               int offset );
```

Arguments:

rcvid

The value returned by *MsgReceive*()* when you received the message.

msg

A pointer to a buffer where the function can store the data.

bytes

The number of bytes that you want to read. These functions don't let you read past the end of the thread's message; they return the number of bytes actually read.

offset

An offset into the thread's send message that indicates where you want to start reading the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgRead()* and *MsgRead_r()* kernel calls read data from a message sent by a thread identified by *rcvid*. The thread being read from must not have been replied to

and will be in the REPLY-blocked state. Any thread in the receiving process is free to read the message.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1755) section for details.

The data transfer occurs immediately and the thread doesn't block. The state of the sending thread doesn't change.

You'll use these functions in these situations:

- A message is sent consisting of a fixed header and a variable amount of data. The header contains the byte count of the data. If the data is large and has to be inserted into one or more buffers (like a filesystem cache), rather than read the data into one large buffer and then copy it into several other buffers, *MsgReceive()* reads only the header, and you can call *MsgRead()* one or more times to read data directly into the required buffer(s).
- A message is received but can't be handled at the present time. At some point in the future, an event will occur that will allow the message to be processed. Rather than saving the message until it can be processed (thus using memory resources), you can use *MsgRead()* to reread the message, during which time the sending thread is still blocked.
- Messages that are larger than available buffer space are received. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use *MsgRead()* to read the message in small pieces, and use *MsgWrite*()* to write the messages in small pieces.

When you're finished using *MsgRead()*, you must use *MsgReply*()* to ready the REPLY-blocked process and complete the message exchange.

Blocking states

None. In the network case, lower priority threads may run.

Native networking

The *MsgRead()* function has increased latency when it's used to communicate across a network — a message pass is involved from the server to the network manager (at least). Depending on the size of the data transfer, the server's `lsm-qnet.so` and the client's `lsm-qnet.so` may need to communicate over the link to read more data bytes from the client.

Returns:

The only difference between the *MsgRead()* and *MsgRead_r()* functions is the way they indicate errors:

MsgRead()

The number of bytes read. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

MsgRead_r()

The number of bytes read. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

If you try to read past the end of the thread's message, the functions return the number of bytes they were actually able to read.

Errors:

EFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ESRCH

The thread indicated by *rcvid* doesn't exist or has had its connection detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgReadv(), MsgReadv_r()

Read data from a message

Synopsis:

```
#include <sys/neutrino.h>

int MsgReadv( int ravid,
              const iov_t* riov,
              int rparts,
              int offset );

int MsgReadv_r( int ravid,
                const iov_t* riov,
                int rparts,
                int offset );
```

Arguments:

ravid

The value returned by *MsgReceive*()* when you received the message.

riov

An array of buffers where the functions can store the data.

rparts

The number of elements in the *riov* array.

offset

An offset into the thread's send message that indicates where you want to start reading the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReadv()* and *MsgReadv_r()* kernel calls read data from a message sent by a thread identified by *ravid*. The thread being read from must not have been replied to and will be in the REPLY-blocked state. Any thread in the receiving process is free to read the message.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1758) section for details.

The data transfer occurs immediately and the thread doesn't block. The state of the sending thread doesn't change.

An attempt to read past the end of the thread's message results in fewer bytes returned than requested.

You'll use these functions in these situations:

- A message is sent consisting of a fixed header and a variable amount of data. The header contains the byte count of the data. If the data is large and has to be inserted into one or more buffers (like a filesystem cache), rather than read the data into one large buffer and then copy it into several other buffers, *MsgReceive()* reads only the header, and you can build a custom *iov_t* list to let *MsgReadv()* read data directly into the required buffers.
- A message is received but can't be handled at the present time. At some point in the future, an event will occur that will allow the message to be processed. Rather than saving the message until it can be processed (thus using memory resources), you can use *MsgReadv()* to reread the message, during which time the sending thread is still blocked.
- Messages that are larger than available buffer space are received. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use *MsgReadv()* to read the message in small pieces, and use *MsgWrite*()* to write the messages in small pieces.

When you're finished using *MsgReadv()*, you must use *MsgReply*()* to ready the REPLY-blocked process and complete the message exchange.

Blocking states

None. In the network case, lower priority threads may run.

Returns:

The only difference between the *MsgReadv()* and *MsgReadv_r()* functions is the way they indicate errors:

MsgReadv()

The number of bytes read. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

MsgReadv_r()

The number of bytes read. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EFAULT**

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ESRCH

The thread indicated by *rcvid* doesn't exist or has had its connection detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgReceive(), MsgReceive_r()

Wait for a message or pulse on a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgReceive( int chid,
               void * msg,
               int bytes,
               struct _msg_info * info );

int MsgReceive_r( int chid,
                 void * msg,
                 int bytes,
                 struct _msg_info * info );
```

Arguments:

chid

The ID of a channel that you established by calling [ChannelCreate\(\)](#) (p. 278).

msg

A pointer to a buffer where the function can store the received data.

bytes

The size of the buffer.

info

NULL, or a pointer to a [_msg_info](#) (p. 1731) structure where the function can store additional information about the message.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReceive()* and *MsgReceive_r()* kernel calls wait for a message or pulse to arrive on the channel identified by *chid*, and store the received data in the buffer pointed to by *msg*.

These functions are identical, except in the way they indicate errors; see the [Returns](#) (p. 1763) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The received data isn't allowed to overflow the receive buffer area provided.



The *msg* buffer *must* be big enough to contain a pulse. If it isn't, the functions indicate an error of EFAULT.

If a message is waiting on the channel when you call *MsgReceive()*, the calling thread doesn't block, and the message is immediately copied. If a message isn't waiting, the calling thread enters the RECEIVE-blocked state until a message arrives.

If multiple messages are sent to a channel without a thread waiting to receive them, the messages are queued in priority order.



The thread's effective priority might change when it receives a message. For more information, see "Priority inheritance and messages" in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

If you pass a non-NULL pointer for *info*, the functions store additional information about the message and the thread that sent it in the `_msg_info` structure that *info* points to. You can get this information later by calling [MsgInfo\(\)](#) (p. 1745).

On success, *MsgReceive()* and *MsgReceive_r()* return:

>0

A message was received; the returned value is a *rcvid* (receive identifier). You'll use the *rcvid* with other *Msg*()* kernel calls to interact with and reply to the sending thread. *MsgReceive()* changes the state of the sending thread to REPLY-blocked when the message is received. When you use *MsgReply*()* to reply to the received message, the sending thread is made ready again. The *rcvid* encodes the sending thread's ID and a local connection ID.

0

A pulse was received; *msg* contains a pulse message of type [_pulse](#) (p. 2440). When a pulse is received, the kernel space allocated to hold it is immediately released. The `_msg_info` structure isn't updated.



Don't reply to a pulse.

Blocking states

State	Meaning
STATE_RECEIVE	There's no message waiting

Native networking

In networked message-passing transactions, the most noticeable impact is on the server. The server receives the client's message from the server's local `lsm-qnet .so`. Note that the receive ID that comes back from `MsgReceive()` will have some differences, but you don't need to worry about the format of the receive ID — just treat it as a “magic cookie.”

When the server unblocks from its `MsgReceive()`, it may or may not have received as much of the message as it would in the local case. This is because of the way that message passing is defined — the client and the server agree on the size of the message transfer area (the transmit parameters passed to `MsgSend()` on the client end) and the size of the message receive area on the server's `MsgReceive()`.

In a local message pass, the kernel would ordinarily limit the size of the transfer to the minimum of both sizes. But in the networked case, the message is received by the client's `lsm-qnet .so` into its own private buffers and then sent via transport to the remote `lsm-qnet .so`. Since the size of the server's receive data area can't be known in advance by the client's `lsm-qnet .so` when the message is sent, only a fixed maximum size (currently 8 KB) message is transferred between the client and the server.

This means, for example, that if the client sends 1 Mbyte of data and the server issues a `MsgReceive()` with a 1-Mbyte data area, then only the number of bytes determined by a network manager would in fact be transferred. The number of bytes transferred to the server is returned via the last parameter to `MsgReceive()` or a call to `MsgInfo()`, specifically the `msglen` member of struct `_msg_info`. The client doesn't notice this, because it's still blocked.

You can use the following code to ensure that the desired number of bytes are received. Note that this is handled for you automatically when you're using the resource manager library:

```
chid = ChannelCreate(_NTO_CHF_SENDER_LEN);
...
rcvid = MsgReceive(chid, msg, nbytes, &info);

/*
 * Doing a network transaction and not all
 * the message was send, so get the rest...
 */
if (rcvid > 0 && info.srcmsglen > info.msglen &&
    info.msglen < nbytes) {
    int n;

    if((n = MsgRead_r(rcvid, (char *) msg + info.msglen,
        nbytes - info.msglen, info.msglen)) < 0) {
        MsgError(rcvid, -n);
        continue;
    }
}
```

```

    info.msglen += n;
}

```

Returns:

The only difference between *MsgReceive()* and *MsgReceive_r()* is the way they indicate errors. On success, both functions return a positive *rcvid* if they received a message, or 0 if they received a pulse.

If an error occurs:

- *MsgReceive()* returns -1 and sets [errno](#) (p. 539).
- *MsgReceive_r()* returns the negative of a value from the Errors section is returned. This function **doesn't** set *errno*.

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender's buffers only when *MsgReceive()* is called, a fault could occur *in the sender* if the sender's buffers are invalid. If a fault occurs when accessing the sender buffers (only) they'll receive an EFAULT and *MsgReceive()* won't unblock.

This error also occurs if the kernel tries to deliver a pulse to the server, but the size of the receive buffer is less than the size of a `struct _pulse`. The pulse is lost in this case.

EINTR

The call was interrupted by a signal.

ESRCH

The channel indicated by *chid* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

The maximum size for a one-part message-pass is $2^{31} - 1$ (SSIZE_MAX).

MsgReceivePulse(), MsgReceivePulse_r()

Receive a pulse on a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgReceivePulse( int chid,
                    void * pulse,
                    int bytes,
                    struct _msg_info * info );

int MsgReceivePulse_r( int chid,
                      void * pulse,
                      int bytes,
                      struct _msg_info * info );
```

Arguments:

chid

The ID of a channel that you established by calling [ChannelCreate\(\)](#) (p. 278).

pulse

A void * pointer to a [struct _pulse](#) (p. 2440) structure where the function can store the received data.



If this buffer isn't big enough to contain a `struct _pulse` structure, you'll get an EFAULT.

bytes

The size of the buffer.

info

The function doesn't update this structure, so you typically pass NULL for this argument.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReceivePulse()* and *MsgReceivePulse_r()* kernel calls wait for a pulse to arrive on the channel identified by *chid* and place the received data in the buffer pointed to by *pulse*. If the buffer is large enough, the number of bytes written to it is the size of a `struct _pulse` structure.

These functions are identical, except in the way they indicate errors; see the [Returns](#) (p. 1766) section for details.

If a pulse is waiting on the channel when you call *MsgReceivePulse()*, the calling thread doesn't block, and the pulse is immediately copied. If a pulse isn't waiting, the calling thread enters the RECEIVE-blocked state until a pulse arrives.

If multiple pulses are sent to a channel without a thread waiting to receive them, the pulses are queued in priority order.

When a thread receives a pulse:

- The kernel space allocated to hold the pulse is immediately released.
- The memory that *pulse* points to contains a pulse message of type `struct _pulse` (p. 2440).
- The thread's effective priority might change. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.



Don't reply to a pulse.

Blocking states

State	Meaning
STATE_RECEIVE	There's no pulse waiting.

Returns:

The only difference between *MsgReceivePulse()* and *MsgReceivePulse_r()* is the way they indicate errors. On success, they both return 0.

If an error occurred:

- *MsgReceivePulse()* returns -1 and sets [errno](#) (p. 539).
- *MsgReceivePulse_r()* returns the negative of a value from the Errors section. This function **doesn't** set *errno*.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffer provided, or the size of the receive buffer is less than the size of a struct `_pulse`. The pulse is lost in this case.

EINTR

The call was interrupted by a signal.

ESRCH

The channel indicated by *chid* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgReceivePulsev(), MsgReceivePulsev_r()

Receive a pulse on a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgReceivePulsev( int chid,
                     const iov_t * piov,
                     int parts,
                     struct _msg_info * info );

int MsgReceivePulsev_r( int chid,
                       const iov_t * piov,
                       int parts,
                       struct _msg_info * info );
```

Arguments:

chid

The ID of a channel that you established by calling [ChannelCreate\(\)](#) (p. 278).

piov

An array of buffers where the function can store the received data. Only the first buffer is used; any additional ones are ignored.



The first buffer of the IOV (input/output vector) *must* be big enough to contain a [struct _pulse](#) (p. 2440) structure. If it isn't, you'll get an EFAULT.

parts

The number of elements in the array.

info

The function doesn't update this structure, so you typically pass NULL for this argument.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReceivePulsev()* and *MsgReceivePulsev_r()* kernel calls wait for a pulse to arrive on the channel identified by *chid* and place the received data in the first element of the array of buffers pointed to by *piov*. If the buffer is large enough, the number of bytes written to it is the size of a `struct _pulse` structure.

These functions are identical, except in the way they indicate errors; see the [Returns](#) (p. 1769) section for details. These functions aren't very useful because the “array of buffers” is really a single element that has to be the size of a `struct _pulse` structure. [MsgReceivePulse\(\)](#) (p. 1765) has a cleaner set of arguments.

If a pulse is waiting on the channel when you call *MsgReceivePulsev()*, the calling thread doesn't block, and the pulse is immediately copied. If a pulse isn't waiting, the calling thread enters the RECEIVE-blocked state until a pulse arrives.

If multiple pulses are sent to a channel without a thread waiting to receive them, the pulses are queued in priority order.

When a thread receives a pulse:

- The kernel space allocated to hold the pulse is immediately released.
- The IOV's first buffer contains a pulse message of type `struct _pulse` (p. 2440).
- The thread's effective priority might change. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.



Don't reply to a pulse.

Blocking states

State	Meaning
STATE_RECEIVE	There's no pulse waiting.

Returns:

The only difference between *MsgReceivePulsev()* and *MsgReceivePulsev_r()* is the way they indicate errors. On success, they both return 0.

If an error occurred:

- *MsgReceivePulsev()* returns -1 and sets [errno](#) (p. 539).
- *MsgReceivePulsev_r()* returns the negative of a value from the Errors section. This function **doesn't** set *errno*.

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided, or the size of the first element in the receive buffer is less than the size of a struct `_pulse`. The pulse is lost in this case.

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds `INT_MAX`.

ESRCH

The channel indicated by *chid* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgReceivev(), MsgReceivev_r()

Wait for a message or pulse on a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgReceivev( int chid,
                 const iov_t * riov,
                 int rparts,
                 struct _msg_info * info );

int MsgReceivev_r( int chid,
                   const iov_t * riov,
                   int rparts,
                   struct _msg_info * info );
```

Arguments:

chid

The ID of a channel that you established by calling [ChannelCreate\(\)](#) (p. 278).

riov

An array of buffers where the function can store the received data.

rparts

The number of elements in the array.

info

NULL, or a pointer to a [_msg_info](#) (p. 1731) structure where the function can store additional information about the message.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReceivev()* and *MsgReceivev_r()* kernel calls wait for a message or pulse to arrive on the channel identified by *chid* and place the received data in the array of buffers pointed to by *riov*.

These functions are identical, except in the way they indicate errors; see the [Returns](#) (p. 1773) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The received data isn't allowed to overflow the receive buffer area provided.



The first buffer of the IOV (input/output vector) *must* be big enough to contain a pulse. If it isn't, the functions indicate an error of EFAULT.

If a message is waiting on the channel when you call *MsgReceivev()*, the calling thread won't block, and the message is immediately copied. If a message isn't waiting, the calling thread enters the RECEIVE-blocked state until a message arrives.

If multiple messages are sent to a channel without a thread waiting to receive them, the messages are queued in priority order.



The thread's effective priority might change when it receives a message. For more information, see "Priority inheritance and messages" in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

If you pass a non-NULL pointer for *info*, the functions store additional information about the message and the thread that sent it in the `_msg_info` structure that *info* points to. You can get this information later by calling [MsgInfo\(\)](#) (p. 1745).

On success, *MsgReceivev()* and *MsgReceivev_r()* return:

>0

A message was received; the value returned is a *rcvid* (receive identifier). You'll use the *rcvid* with other *Msg*()* kernel calls to interact with and reply to the sending thread. *MsgReceivev()* changes the state of the sending thread to REPLY-blocked when the message is received. When you use *MsgReply*()* to reply to the received message, the sending thread is made ready again. The *rcvid* encodes the sending thread's ID and a local connection ID.

0

A pulse was received; the IOV's first buffer contains a pulse message of type [_pulse](#) (p. 2440). When a pulse is received, the kernel space allocated to hold it is immediately released. The `_msg_info` structure isn't updated.



Don't reply to a pulse.

Blocking states

State	Meaning
STATE_RECEIVE	There's no message waiting.

Returns:

The only difference between *MsgReceivev()* and *MsgReceivev_r()* is the way they indicate errors. On success, both functions return a positive *rcvid* if they received a message, or 0 if they received a pulse.

If an error occurs:

- *MsgReceivev()* returns -1 and sets [errno](#) (p. 539).
- *MsgReceivev_r()* returns the negative of a value from the Errors section. This function **doesn't** set *errno*.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender's buffers only when *MsgReceivev()* is called, a fault could occur *in the sender* if the sender's buffers are invalid. If a fault occurs when accessing the sender buffers (only) they'll receive an EFAULT and *MsgReceivev()* won't unblock.

This error also occurs if the kernel tries to deliver a pulse to the server, but the size of the receive buffer is less than the size of a `struct _pulse`. The pulse is lost in this case.

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds INT_MAX.

ESRCH

The channel indicated by *chid* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgReply(), MsgReply_r()

Reply with a message

Synopsis:

```
#include <sys/neutrino.h>

int MsgReply( int ravid,
              int status,
              const void* msg,
              int size );

int MsgReply_r( int ravid,
                int status,
                const void* msg,
                int size );
```

Arguments:***ravid***

The receive ID that *MsgReceive*()* returned when you received the message.

status

The status to use when unblocking the client's *MsgSend*()* call in the *ravid* thread.



The *MsgSend*_r()* functions return negative *errno* values to indicate failure, so you shouldn't pass a negative value for the *status* to *MsgReply*()*, because the *MsgSend*_r()* functions could interpret it as an error code. If you wish to indicate an error, call [MsgError\(\)](#) (p. 1743).

msg

A pointer to a buffer that contains the message that you want to reply with.

size

The size of the message, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReply()* and *MsgReply_r()* kernel calls reply with a message to the thread identified by *rcvid*. The thread being replied to must be in the REPLY-blocked state. Any thread in the receiving process is free to reply to the message, however, it may be replied to only once for each receive.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1776) section for details.

The *MsgSend*()* in the *rcvid* thread unblocks with a return value of *status*.

The number of bytes transferred is the minimum of that specified by both the replier and the sender. The reply data isn't allowed to overflow the reply buffer area provided by the sender.

The data transfer occurs immediately, and the replying task doesn't block. There's no need to reply to received messages in any particular order, but you must eventually reply to each message to allow the sending thread(s) to continue execution.

Blocking states

None. In the network case, lower priority threads may run.

Native networking

The *MsgReply()* function has increased latency when it's used to communicate across a network — the server is now writing data to its local `lsm-qnet.so`, which may need to communicate with the client's `lsm-qnet.so` to actually transfer the data.

Returns:

The only difference between the *MsgReply()* and *MsgReply_r()* functions is the way they indicate errors:

MsgReply()

If an error occurs, `-1` is returned and [errno](#) (p. 539) is set.

MsgReply_r()

This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EFAULT

A fault occurred in the sender's address space when a server tried to access the sender's return message buffers.

ENOREMOTE

The reply has to go across the network, and Qnet isn't running.

ESRCH

The thread indicated by *rcvid* doesn't exist, or is no longer REPLY-blocked on the channel, or the connection is detached.

ESRVRFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The maximum size for a one-part message-pass is $2^{31} - 1$ (SSIZE_MAX).

MsgReplyv(), MsgReplyv_r()

Reply with a message

Synopsis:

```
#include <sys/neutrino.h>

int MsgReplyv( int rcvid,
               int status,
               const iov_t* riov,
               int rparts );

int MsgReplyv_r( int rcvid,
                 int status,
                 const iov_t* riov,
                 int rparts );
```

Arguments:

rcvid

The receive ID that *MsgReceive*()* returned when you received the message.

status

The status to use when unblocking the client's *MsgSend*()* call in the *rcvid* thread.



The *MsgSend*_r()* functions return negative *errno* values to indicate failure, so you shouldn't pass a negative value for the *status* to *MsgReply*()*, because the *MsgSend*_r()* functions could interpret it as an error code. If you wish to indicate an error, call [MsgError\(\)](#) (p. 1743).

riov

An array of buffers that contains the message that you want to reply with.

rparts

The number of elements in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgReplyv()* and *MsgReplyv_r()* kernel calls reply with a message to the thread identified by *rcvid*. The thread being replied to must be in the REPLY-blocked state. Any thread in the receiving process is free to reply to the message, however, it may be replied to only once for each receive.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1779) section for details.

The *MsgSend*()* in the *rcvid* thread unblocks with a return value of *status*.

The data is taken from the array of message buffers pointed to by *riov*. The number of elements in this array is given by *rparts*. The size of the message is the sum of the sizes of each buffer.

The number of bytes transferred is the minimum of that specified by both the replier and the sender. The reply data isn't allowed to overflow the reply buffer area provided by the sender.

The data transfer occurs immediately, and the replying task doesn't block. There's no need to reply to received messages in any particular order, but you must eventually reply to each message to allow the sending thread(s) to continue execution.

It's quite common to reply with two-part messages consisting of a fixed header and a buffer of data. The *MsgReplyv()* function gathers the data from the buffer list into a logical contiguous message and transfers it to the sender's reply buffer(s). The sender doesn't need to specify the same number or size of buffers. The data is laid down filling each buffer as required. The filesystem, for example, builds a reply list pointing into its cache in order to reply with what appears to be one contiguous piece of data.

Blocking states

None. In the network case, lower priority threads may run.

Returns:

The only difference between the *MsgReplyv()* and *MsgReplyv_r()* functions is the way they indicate errors:

MsgReplyv()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

MsgReplyv_r()

This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EFAULT**

A fault occurred in the sender's address space when a server tried to access the sender's return message buffers.

ENOREMOTE

The reply has to go across the network, and Qnet isn't running.

ESRCH

The thread indicated by *rcvid* doesn't exist, or is no longer REPLY-blocked on the channel, or the connection is detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgSend(), MsgSend_r()

Send a message to a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgSend( int coid,
             const void* smsg,
             int sbytes,
             void* rmsg,
             int rbytes );

int MsgSend_r( int coid,
               const void* smsg,
               int sbytes,
               void* rmsg,
               int rbytes );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

smsg

A pointer to a buffer that contains the message that you want to send.

sbytes

The number of bytes to send.

rmsg

A pointer to a buffer where the reply can be stored.

rbytes

The size of the reply buffer, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSend()* and *MsgSend_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1783) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

MsgSend() is a cancellation point for the *ThreadCancel()* kernel call; *MsgSendnc()* isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Native networking

When a client sends a message to a remote server, the client is effectively sending the message via its local microkernel; the network manager does the actual “work.” The local network manager negotiates with the remote network manager and causes the message to be delivered there. However, the remote manager is the one that actually delivers the message to the server.

This message transfer from the remote manager to the server is accomplished via a special nonblocking message pass.

The only impact on the client is the latency of the message-passing operations. This is purely a function of the network link speed and the overhead associated with the protocol (i.e. `lsm-qnet.so` for native networking) that `io-pkt*` uses.

The client still remains blocked in its `MsgSend()`, and unblocks only on account of a signal, a kernel timeout, or the completion of its function.

Returns:

The only difference between the `MsgSend()` and `MsgSend_r()` functions is the way they indicate errors:

MsgSend()

Success

The value of *status* from `MsgReply*()`.

-1

An error occurred ([errno](#) (p. 539) is set), or the server called `MsgError*()` (*errno* is set to the error value passed to `MsgError()`).

MsgSend_r()

Success

The value of *status* from `MsgReply*()`.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (*errno* is *NOT* set, the value is the negative of the error value passed to `MsgError()`).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The maximum size for a one-part message-pass is $2^{31} - 1$ (SSIZE_MAX).

MsgSendnc(), MsgSendnc_r()

Send a message to a channel (non-cancellation point)

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendnc( int coid,
               const void* smsg,
               int sbytes,
               void* rmsg,
               int rbytes );

int MsgSendnc_r( int coid,
                 const void* smsg,
                 int sbytes,
                 void* rmsg,
                 int rbytes );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

smsg

A pointer to a buffer that contains the message that you want to send.

sbytes

The number of bytes to send.

rmsg

A pointer to a buffer where the reply can be stored.

rbytes

The size of the reply buffer, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendnc()* and *MsgSendnc_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1786) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

MsgSend() is a cancellation point for the *ThreadCancel()* kernel call; *MsgSendnc()* isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendnc()* and *MsgSendnc_r()* functions is the way they indicate errors:

MsgSendnc()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* is set), or the server called *MsgError*()* (*errno* (p. 539) is set to the error value passed to *MsgError()*).

MsgSendnc_r()

Success

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The maximum size for a one-part message-pass is $2^{31} - 1$ (SSIZE_MAX).

MsgSendPulse(), MsgSendPulse_r()

Send a pulse to a process

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendPulse ( int coid,
                  int priority,
                  int code,
                  int value );

int MsgSendPulse_r ( int coid,
                    int priority,
                    int code,
                    int value );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

priority

The priority to use for the pulse, or -1 to use the priority of the calling thread. The priority must be within the range of valid priorities, which you can determine by calling [sched_get_priority_min\(\)](#) (p. 2727) and [sched_get_priority_max\(\)](#) (p. 2725).

code

The 8-bit pulse code.

Although *code* can be any 8-bit signed value, you should avoid *code* values less than zero, in order to avoid conflict with pulse codes generated by the kernel or a QNX Neutrino manager. These codes all start with `_PULSE_CODE_` and are defined in `<sys/neutrino.h>`; for more information, see the documentation for the [_pulse](#) (p. 2440) structure. A safe range of pulse values is `_PULSE_CODE_MINAVAIL` through `_PULSE_CODE_MAXAVAIL`.

value

The 32-bit pulse value.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendPulse()* and *MsgSendPulse_r()* kernel calls send a short, nonblocking message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1791) section for details.

You can send a pulse to a process if the sending process's real or effective user ID either:

- matches the real or effective user ID of the receiving process

Or:

- equals zero.

This permission checking is identical to that used by [kill\(\)](#) (p. 1494).

You can use *MsgSendPulse()* for many purposes; however, due to the small payload of data, you shouldn't use it for transmitting large amounts of bulk data by sending a great number of pulses.

Pulses are queued for the receiving process in the system, using a dynamic pool of memory objects. If pulses are generated faster than they can be consumed by the receiver, then over a period of time the system queue for the pulses could reach a low memory condition. If there's no memory available for the pulse to be queued in the system, the kernel fails the pulse request with an error of `EAGAIN`. If the priority, code and value don't change, the kernel compresses the pulses by storing an 8-bit count with an already queued pulse.

-
- In order to send a pulse to a channel that was created by a different process with a different user ID, your process must have the `PROCGR_AID_CONNECTION` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- The receiving thread's effective priority might change when you send a pulse to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.
-

When you receive a pulse via the *MsgReceive*()* kernel call, the *rcvid* returned is zero. This indicates to the receiver that it's a pulse and, unlike a message, shouldn't be replied to using *MsgReply*()*.



In a client/server design, *MsgDeliverEvent()* (p. 1736) is typically used in the server, and *MsgSendPulse()* in the client.

Blocking states

None for the local case. In the network case:

STATE_NET_SEND

The calling thread is blocked while the pulse is delivered to the local Qnet manager, which transmits it to the remote Qnet. The remote Qnet does a *MsgSendPulse()* on behalf of the caller, and the return value is transmitted back, at which point the calling thread is unblocked.

Native networking

You can use *MsgSendPulse()* to send pulses across the network.

Returns:

The only difference between the *MsgSendPulse()* and *MsgSendPulse_r()* functions is the way they indicate errors:

MsgSendPulse()

If an error occurs, -1 is returned and *errno* (p. 539) is set. Any other value returned indicates success.

MsgSendPulse_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EAGAIN

The kernel had insufficient resources to enqueue the pulse.

EBADF

The connection indicated by *coid* is no longer connected to a channel or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINVAL

The specified priority is invalid (e.g. 0).

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRVFAULT

A fault occurred in the server's address space when the kernel tried to write the pulse message to the server's receive message buffer.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

If the server faults on delivery, the pulse is either lost or an error is returned.

MsgSendsv(), MsgSendsv_r()

Send a message to a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendsv( int coid,
               const void* smsg,
               int sbytes,
               const iov_t* riov,
               int rparts );

int MsgSendsv_r( int coid,
                 const void* smsg,
                 int sbytes,
                 const iov_t* riov,
                 int rparts );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

smsg

A pointer to a buffer that contains the message that you want to send.

sbytes

The number of bytes to send.

riov

An array of buffers where the reply can be stored.

rparts

The number of elements in the *riov* array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendsv()* and *MsgSendsv_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1794) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

MsgSendsv() is a cancellation point for the *ThreadCancel()* kernel call; [MsgSendsvnc\(\)](#) (p. 1797) isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendsv()* and *MsgSendsv_r()* functions is the way they indicate errors:

MsgSendsv()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* (p. 539) is set), or the server called *MsgError*()* (*errno* is set to the error value passed to *MsgError()*).

MsgSendsv_r()

Success

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds *INT_MAX*.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgSendsvnc(), MsgSendsvnc_r()

Send a message to a channel (non-cancellation point)

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendsvnc( int coid,
                 const void* smsg,
                 int sbytes,
                 const iov_t* riov,
                 int rparts );

int MsgSendsvnc_r( int coid,
                   const void* smsg,
                   int sbytes,
                   const iov_t* riov,
                   int rparts );
```

Arguments:***coid***

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

smsg

A pointer to a buffer that contains the message that you want to send.

sbytes

The number of bytes to send.

riov

An array of buffers where the reply can be stored.

rparts

The number of elements in the *riov* array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendsvnc()* and *MsgSendsvnc_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1798) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

[MsgSendsv\(\)](#) (p. 1793) is a cancellation point for the *ThreadCancel()* kernel call; *MsgSendsvnc()* isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendsvnc()* and *MsgSendsvnc_r()* functions is the way they indicate errors:

MsgSendsvnc()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* (p. 539) is set), or the server called *MsgError*()* (*errno* is set to the error value passed to *MsgError()*).

MsgSendsvnc_r()

Success

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds *INT_MAX*.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgSendv(), MsgSendv_r()

Send a message to a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendv( int coid,
              const iov_t* siov,
              int sparts,
              const iov_t* riov,
              int rparts );

int MsgSendv_r( int coid,
                const iov_t* siov,
                int sparts,
                const iov_t* riov,
                int rparts );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

siov

An array of buffers that contains the message that you want to send.

sparts

The number of elements in the *siov* array.

riov

An array of buffers where the reply can be stored.

rparts

The number of elements in the *riov* array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendv()* and *MsgSendv_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1803) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).

It's quite common to send two-part messages consisting of a fixed header and a buffer of data. The *MsgSendv()* function gathers the data from the send list into a logically contiguous message and transfers it to the receiver. The receiver doesn't need to specify the same number or size of buffers. The data is laid down filling each entry as required. The same applies to the replied data.



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

MsgSendv() is a cancellation point for the *ThreadCancel()* kernel call; [MsgSendvnc\(\)](#) (p. 1805) isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendv()* and *MsgSendv_r()* functions is the way they indicate errors:

MsgSendv()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* (p. 539) is set), or the server called *MsgError*()* (*errno* is set to the error value passed to *MsgError()*).

MsgSendv_r()**Success**

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:**EBADF**

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

E_OVERFLOW

The sum of the IOV lengths exceeds INT_MAX.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgSendvnc(), MsgSendvnc_r()

Send a message to a channel (non-cancellation point)

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendvnc( int coid,
                const iov_t* siov,
                int sparts,
                const iov_t* riov,
                int rparts );

int MsgSendvnc_r( int coid,
                  const iov_t* siov,
                  int sparts,
                  const iov_t* riov,
                  int rparts );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

siov

An array of buffers that contains the message that you want to send.

sparts

The number of elements in the *siov* array.

riov

An array of buffers where the reply can be stored.

rparts

The number of elements in the *riov* array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendvnc()* and *MsgSendvnc_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1806) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

[MsgSendv\(\)](#) (p. 1801) is a cancellation point for the [ThreadCancel\(\)](#) (p. 3399) kernel call; *MsgSendvnc()* isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendvnc()* and *MsgSendvnc_r()* functions is the way they indicate errors:

MsgSendvnc()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* (p. 539) is set), or the server called *MsgError*()* (*errno* is set to the error value passed to *MsgError()*).

MsgSendvnc_r()

Success

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds *INT_MAX*.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgSendvs(), MsgSendvs_r()

Send a message to a channel

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendvs( int coid,
               const iov_t* siov,
               int sparts,
               void* rmsg,
               int rbytes );

int MsgSendvs_r( int coid,
                 const iov_t* siov,
                 int sparts,
                 void* rmsg,
                 int rbytes );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

siov

An array of buffers that contains the message that you want to send.

sparts

The number of elements in the *siov* array.

rmsg

A pointer to a buffer where the reply can be stored.

rbytes

The size of the reply buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendvs()* and *MsgSendvs_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1810) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

MsgSendvs() is a cancellation point for the [ThreadCancel\(\)](#) (p. 3399) kernel call; [MsgSendvsnc\(\)](#) (p. 1813) isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendvs()* and *MsgSendvs_r()* functions is the way they indicate errors:

MsgSendvs()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* (p. 539) is set), or the server called *MsgError*()* (*errno* is set to the error value passed to *MsgError()*).

MsgSendvs_r()

Success

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds *INT_MAX*.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgSendvsnc(), MsgSendvsnc_r()

Send a message to a channel (non-cancellation point)

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendvsnc( int coid,
                 const iov_t* siov,
                 int sparts,
                 void* rmsg,
                 int rbytes );

int MsgSendvsnc_r( int coid,
                  const iov_t* siov,
                  int sparts,
                  void* rmsg,
                  int rbytes );
```

Arguments:

coid

The ID of the channel to send the message on, which you've established by calling [ConnectAttach\(\)](#) (p. 351) or [name_open\(\)](#) (p. 1852).

siov

An array of buffers that contains the message that you want to send.

sparts

The number of elements in the *siov* array.

rmsg

A pointer to a buffer where the reply can be stored.

rbytes

The size of the reply buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgSendvsnr()* and *MsgSendvsnr_r()* kernel calls send a message to a process's channel identified by *coid*.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1814) section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn't allowed to overflow the receive buffer area provided by the receiver. The reply data isn't allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that's RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes [REPLY-blocked](#) (p. ?). If there are no waiting threads on the channel, the sending thread becomes [SEND-blocked](#) (p. ?) and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn't occur until a receiving thread receives on the channel. At this point, the sending thread becomes [REPLY-blocked](#) (p. ?).



The receiving thread's effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

[MsgSendvsnr\(\)](#) (p. 1809) is a cancellation point for the [ThreadCancel\(\)](#) (p. 3399) kernel call; *MsgSendvsnr()* isn't.

Blocking states**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between the *MsgSendvsnr()* and *MsgSendvsnr_r()* functions is the way they indicate errors:

MsgSendvsnr()**Success**

The value of *status* from *MsgReply*()*.

-1

An error occurred (*errno* (p. 539) is set), or the server called *MsgError*()* (*errno* is set to the error value passed to *MsgError()*).

MsgSendvsnc_r()

Success

The value of *status* from *MsgReply*()*.

negative value

An error occurred (*errno* is *NOT* set, the value is the negative of a value from the Errors section), or the server called *MsgError*()* (*errno* is *NOT* set, the value is the negative of the error value passed to *MsgError()*).

Errors:

EBADF

The connection indicated by *coid* is no longer connected to a channel, or the connection indicated by *coid* doesn't exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT

A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EHOSTDOWN

The host is down (e.g. a send across Qnet failed).

EHOSTUNREACH

Unable to communicate with remote node (e.g. across Qnet).

EINTR

The call was interrupted by a signal.

EOVERFLOW

The sum of the IOV lengths exceeds *INT_MAX*.

ESRCH

The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT

A fault occurred in a server's address space when the kernel tried to access the server's message buffers.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgVerifyEvent(), MsgVerifyEvent_r()

Check the validity of a receive ID and an event configuration

Synopsis:

```
#include <sys/neutrino.h>

int MsgVerifyEvent( int rcvid,
                   const struct sigevent event );

int MsgVerifyEvent_r( int rcvid,
                     const struct sigevent event );
```

Arguments:

rcvid

The receive ID that you want to check.

event

A pointer to a [sigevent](#) (p. 2961) structure that contains the event you want to check.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgVerifyEvent()* and *MsgVerifyEvent_r()* kernel calls check the validity of the receive ID *rcvid*, and the *event* configuration. You can use these functions to verify that an event is well-formed by a client (pass a *rcvid* of 0), and by a server (pass a *rcvid* of the target thread).

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1818) section for details.

Blocking states

These calls don't block.

Returns:

The only difference between the *MsgVerifyEvent()* and *MsgVerifyEvent_r()* functions is the way they indicate errors:

MsgVerifyEvent()

If an error occurs, -1 is returned and *errno* (p. 539) is set.

MsgVerifyEvent_r()

This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EBADF**

The channel for the pulse delivery doesn't exist.

EINVAL

Invalid *event* structure.

ESRCH

The connection for the pulse doesn't exist.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgWrite(), MsgWrite_r()

Write a reply

Synopsis:

```
#include <sys/neutrino.h>

int MsgWrite( int ravid,
              const void* msg,
              int size,
              int offset );

int MsgWrite_r( int ravid,
                const void* msg,
                int size,
                int offset );
```

Arguments:

ravid

The value returned by *MsgReceive*()* when you received the message.

msg

A pointer to a buffer that contains the data you want to write.

size

The number of bytes that you want to write. These functions don't let you write past the end of the sender's buffer; they return the number of bytes actually written.

offset

An offset into the sender's buffer that indicates where you want to start writing the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *MsgWrite()* and *MsgWrite_r()* kernel calls write data to the reply buffer of a thread identified by *rcvid*. The thread being written to must be in the REPLY-blocked state. Any thread in the receiving process is free to write to the reply message.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1821) section for details.

You use this function in one of these situations:

- The data arrives over time and is quite large. Rather than buffer all the data, you can use *MsgWrite()* to write it into the destination thread's reply message buffer, as it arrives.
- Messages are received that are larger than available buffer space. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use *MsgRead*()* to read messages in small pieces, and use *MsgWrite()* to write messages in small pieces.

To complete a message exchange, you must call *MsgReply*()*. The reply doesn't need to contain any data. If it does contain data, then the data is always written at offset zero in the destination thread's reply message buffer. This is a convenient way of writing the header once all of the information has been gathered.

A single call to *MsgReply*()* is always more efficient than calls to *MsgWrite()* followed by a call to *MsgReply*()*.

Blocking states

None. In the network case, lower priority threads may run.

Native networking

The *MsgWrite()* function has increased latency when you use it to communicate across a network — the server is now writing data to its local `lsm-qnet.so`, which may need to communicate with the client's `lsm-qnet.so` to actually transfer the data. The server's *MsgWrite()* call effectively sends a message to the server's `lsm-qnet.so` to initiate this data transfer.

But since the server's `lsm-qnet.so` has no way to determine the size of the client's receive data area, the number of bytes reported as having been transferred by the server during its *MsgWrite()* call *might not be accurate* — the reported number will instead reflect the number of bytes transferred by the server to its `lsm-qnet.so`.

The message is buffered in the server side's `lsm-qnet.so` until the client replies, in order to reduce the number of network transactions.

If you want to determine the size of the sender's reply buffer, set the `_NTO_CHF_REPLY_LEN` when you call [ChannelCreate\(\)](#) (p. 278).

Returns:

The only difference between the *MsgWrite()* and *MsgWrite_r()* functions is the way they indicate errors:

MsgWrite()

The number of bytes written. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

MsgWrite_r()

The number of bytes written. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EFAULT**

A fault occurred in the sender's address space when the kernel tried to access the sender's return message buffer.

ESRCH

The thread indicated by *rcvid* doesn't exist or its connection was detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MsgWritev(), MsgWritev_r()

Write a reply

Synopsis:

```
#include <sys/neutrino.h>

int MsgWritev( int rcvid,
               const iov_t* iov,
               int parts,
               int offset );

int MsgWritev_r( int rcvid,
                 const iov_t* iov,
                 int parts,
                 int offset );
```

Arguments:

rcvid

The value returned by *MsgReceive*()* when you received the message.

iov

An array of buffers that contains the data you want to write.

parts

The number of elements in the array. These functions don't let you write past the end of the sender's buffer; they return the number of bytes actually written.

offset

An offset into the sender's buffer that indicates where you want to start writing the data.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *MsgWritev()* and *MsgWritev_r()* kernel calls write data to the reply buffer of a thread identified by *rcvid*. The thread being written to must be in the REPLY-blocked state. Any thread in the receiving process is free to write to the reply message.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 1823) section for details.

The data transfer occurs immediately and your thread doesn't block. The state of the sending thread doesn't change.

You'll use this function in one of these situations:

- The data arrives over time and is quite large. Rather than buffer all the data, you can use *MsgWritev()* to write it into the destination thread's reply message buffer, as it arrives.
- Messages are received that are larger than available buffer space. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use *MsgRead*()* to read messages in small pieces, and use *MsgWritev()* to write messages in small pieces.

To complete a message exchange, you must call *MsgReply*()*. The reply doesn't need to contain any data. If it does contain data, then the data is always written at offset zero in the destination thread's reply message buffer. This is a convenient way of writing the header once all of the information has been gathered.

A single call to *MsgReply*()* is always more efficient than calls to *MsgWritev()* followed by a call to *MsgReply*()*.

Blocking states

None. In the network case, lower priority threads may run.

Returns:

The only difference between the *MsgWritev()* and *MsgWritev_r()* functions is the way they indicate errors:

MsgWritev()

The number of bytes written. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

MsgWritev_r()

The number of bytes written. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EFAULT

A fault occurred in the sender's address space when the kernel tried to access the sender's return message buffer.

ESRCH

The thread indicated by *rcvid* doesn't exist or its connection was detached.

ESRVRFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

msync()

Synchronize memory with physical storage

Synopsis:

```
#include <sys/mman.h>

int msync( void * addr,
           size_t len,
           int flags );
```

Arguments:

addr

The beginning of the range of addresses that you want to synchronize.

len

The length of the range of addresses, in bytes.

flags

A bitwise inclusive OR of one or more of the following flags:

- **MS_ASYNC** — perform asynchronous writes. The function returns immediately once all the write operations are initiated or queued for servicing.
- **MS_CACHE_ONLY** (QNX Neutrino extension; QNX Neutrino Core OS 6.3.2 or later) — now that QNX Neutrino includes POSIX support for memory-mapped files, *msync()* performs its intended operation of writing changes in the mapped memory back to the underlying file. Specify this bit if you want to perform the QNX Neutrino extension of having the function flush or invalidate the data cache (via the **MS_ASYNC**, **MS_SYNC**, or **MS_INVALIDATE** flags), instead of the POSIX standard behavior.
- **MS_INVALIDATE** — invalidate cached data. Invalidates all cached copies of mapped data that are inconsistent with the permanent storage locations such that subsequent references obtain data that was consistent with the permanent storage locations sometime between the call to *msync()* and the first subsequent memory reference to the data.
- **MS_INVALIDATE_ICACHE** (QNX Neutrino extension) — if you're dynamically modifying code, use this flag to make sure that the new code is what will be executed.

- `MS_SYNC` — perform synchronous writes. The function doesn't return until all write operations are completed as defined for synchronized I/O data integrity completion.



You can specify at most one of `MS_ASYNC` and `MS_SYNC`, not both.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `msync()` function writes all modified data to permanent storage locations, if any, in those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. The `msync()` function is used with memory mapped files. If no such storage exists, `msync()` need not have any effect. If requested, the `msync()` function then invalidates cached copies of data.

For mappings to files, this function ensures that all write operations are completed as defined for synchronized I/O data integrity completion.



Mappings to files aren't implemented on all filesystems.

If you call `msync()` on `MAP_PRIVATE` mappings, any modified data isn't written to the underlying object and doesn't cause such data to be made visible to other processes.

The behavior of `msync()` is unspecified if the mapping wasn't established by a call to [mmap\(\)](#) (p. 1656).

If `msync()` causes any write to a file, the file's `st_ctime` and `st_mtime` fields are marked for update.

Returns:

0

Success

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBUSY**

Some or all of the addresses in the range starting at *addr* and continuing for *len* bytes are locked, and `MS_INVALIDATE` is specified.

EINTR

The call was interrupted by a signal.

EINVAL

Invalid *flags* value.

ENOMEM

The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that aren't mapped.

Examples:

See the entry for [mmap\(\)](#) (p. 1656).

Classification:

[POSIX 1003.1 MF SIO](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

`MS_INVALIDATE_ICACHE` and `MS_CACHE_ONLY` are QNX Neutrino extensions. The behavior of this function changed in the QNX Neutrino Core OS 6.3.2; to get the previous behavior of just data-cache operations, add the `MS_CACHE_ONLY` flag to the call in addition to `MS_ASYNC`, `MS_SYNC`, or `MS_INVALIDATE`.

munlock()

Unlock the range of process address space already allocated

Synopsis:

```
#include <sys/mman.h>

int munlock(const void * addr,
            size_t len);
```

Arguments:

addr

The starting address for the range of process address space.

len

The amount of memory to unlock, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *munlock()* function unlocks the specified range of address space, which was locked by a call to *mlock()* (p. 1651) or *mlockall()* (p. 1653). Memory locks don't stack; the memory is unlocked, no matter how many times the same process locked it.



The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.

The *munlock()* function doesn't affect those pages that have been mapped and locked into the address spaces of other processes.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**ENOMEM**

Some or all of the address range specified by the *addr* and *len* arguments doesn't correspond to valid mapped pages in the address space of the process.

Classification:

[POSIX 1003.1 MLR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

munlockall()

Unlock a process's address space

Synopsis:

```
#include <sys/mman.h>

int munlockall( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *munlockall()* function unlocks all the currently mapped pages of the address space of the calling process. You can lock memory by calling [mlock\(\)](#) (p. 1651) or [mlockall\(\)](#) (p. 1653).

Any pages that become mapped into the address space of the process after it calls *munlockall()* aren't locked, unless there's an intervening call to *mlockall()* specifying `MCL_FUTURE` or a subsequent call to *mlockall()* specifying `MCL_CURRENT`.

Calling this function doesn't affect any memory locks set by other processes on the pages.



The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.

Returns:

0.

Classification:

[POSIX 1003.1 ML](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

munmap()

Unmap previously mapped addresses

Synopsis:

```
#include <sys/mman.h>

int munmap( void * addr,
            size_t len );
```

Arguments:

addr

The beginning of the range of addresses that you want to unmap.

len

The length of the range of addresses, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *munmap()* function removes any mappings for pages in the address range starting at *addr* and continuing for *len* bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process.

If there are no mappings in the specified address range, then *munmap()* has no effect.

Returns:

0

Success.

-1

Failure; [errno](#) (p. 539) is set.

Errors:

EINVAL

The addresses in the specified range are outside the range allowed for the address space of a process.

EINTR

The call was interrupted by a signal.

ENOMEM

The memory manager fails to allocate memory to handle a user's *munmap()* request. This allocation of memory is necessary for internal structures to represent the new state of mapped memory.

ENOSYS

The function *munmap()* isn't supported by this implementation.

Classification:

[*POSIX 1003.1 MFISHMITYM*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

munmap_device_io()

Free access to a device's registers

Synopsis:

```
#include <sys/mman.h>

int munmap_device_io( uintptr_t io,
                     size_t len );
```

Arguments:

io

The address of the area that you want to unmap.

len

The number of bytes of device I/O memory that you want to unmap.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *munmap_device_io()* unmaps *len* bytes of device I/O memory at *io* (that was previously mapped with [mmap_device_io\(\)](#) (p. 1666)).

Returns:

-1

An error occurred ([errno](#) (p. 539) is set).

Any other value

Successful unmapping.

Errors:

EINVAL

The addresses in the specified range are outside the range allowed for the address space of a process.

ENOSYS

The function *munmap()* isn't supported by this implementation.

ENXIO

The address from *io* for *len* bytes is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

munmap_device_memory()

Unmap previously mapped addresses

Synopsis:

```
#include <sys/mman.h>

int munmap_device_memory( void * addr,
                          size_t len );
```

Arguments:

addr

The beginning of the range of addresses that you want to unmap.

len

The length of the range of addresses, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `munmap_device_memory()` function is essentially the same as [munmap\(\)](#) (p. 1832). It removes any mappings for pages in the address range starting at `addr` and continuing for `len` bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process.

If there are no mappings in the specified address range, then [munmap\(\)](#) (p. 1832) has no effect.

This function is the complement of [mmap_device_memory\(\)](#) (p. 1668).

Returns:

-1

An error occurred ([errno](#) (p. 539) is set).

Any other value

Success.

Errors:**EINVAL**

The addresses in the specified range are outside the range allowed for the address space of a process.

ENOSYS

The [munmap\(\)](#) (p. 1832) function isn't supported by this implementation.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

munmap_flags()

Unmap previously mapped addresses, exercising more control than possible with `munmap()`

Synopsis:

```
#include <sys/mman.h>

int munmap_flags( void * addr,
                  size_t len,
                  unsigned flags );
```

Arguments:

addr

The beginning of the range of addresses that you want to unmap.

len

The length of the range of addresses, in bytes.

flags

0, or a combination of the following bits:

- `UNMAP_INIT_REQUIRED`
- `UNMAP_INIT_OPTIONAL`

For more information, see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `munmap_flags()` function removes any mappings for pages in the address range starting at *addr* and continuing for *len* bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process.

If there are no mappings in the specified address range, then `munmap_flags()` has no effect.



This function was added in the QNX Neutrino Core OS 6.3.2.

If you specify 0 for the *flags*, *munmap_flags()* is the same as *munmap()* (p. 1832). The flags change the way *munmap_flags()* behaves, as follows:

UNMAP_INIT_REQUIRED

POSIX initialization of the page to all zeros is required the next time the underlying physical memory is allocated.

UNMAP_INIT_OPTIONAL

Initialization of the underlying physical memory to zeros on its next allocation is optional. If you specify this flag, and you specify *MAP_NOINIT* when you next call *mmap()* (p. 1656) for this physical memory, the POSIX requirement that the memory be zeroed is relaxed.

The memory-manager configuration (-m) option for *procnto* affects the default behavior of *munmap_flags()* (unless overridden by the *flags* argument), as detailed below:

-mi

Act as if *UNMAP_INIT_REQUIRED* were specified (the default).

-m~i

Act as if *UNMAP_INIT_OPTIONAL* were specified.

Returns:

0

Success.

-1

Failure; *errno* (p. 539) is set.

Errors:

EINVAL

The addresses in the specified range are outside the range allowed for the address space of a process.

ENOSYS

The function *munmap_flags()* isn't supported by this implementation.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Chapter 16

N

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<code>_NTO_TRACE_GETEVENT_C()</code>	“Macros for use with the event header (p. 3517)” in the entry for <code>TraceEvent()</code>
<code>_NTO_TRACE_GETEVENT()</code>	“Macros for use with the event header (p. 3517)” in the entry for <code>TraceEvent()</code>
<code>_NTO_TRACE_GETCPU()</code>	“Macros for use with the event header (p. 3517)” in the entry for <code>TraceEvent()</code>
<code>_NTO_TRACE_SETEVENT_C()</code>	“Macros for use with the event header (p. 3517)” in the entry for <code>TraceEvent()</code>
<code>_NTO_TRACE_SETEVENT()</code>	“Macros for use with the event header (p. 3517)” in the entry for <code>TraceEvent()</code>

The following have been deprecated:

Instead of:	Use:
<code>nanosleep_abs()</code>	clock_nanosleep() (p. 315)
<code>nanosleep_rel()</code>	clock_nanosleep() (p. 315)

Note the following:

- [nice\(\)](#) (p. 1892) has no effect in QNX Neutrino; use [pthread_setschedparam\(\)](#) (p. 2400) or [SchedSet\(\)](#) (p. 2780)

This chapter describes the API components whose names begin with “N”.

name_attach()

Register a name in the namespace and create a channel

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

name_attach_t * name_attach( dispatch_t * dpp,
                             const char * path,
                             unsigned flags );
```

Arguments:

dpp

NULL, or a dispatch handle returned by a successful call to [dispatch_create\(\)](#) (p. 433) or [dispatch_create_channel\(\)](#) (p. 436).

path

The path that you want to register under `/dev/name/[local|global]/`. This name shouldn't contain any path components consisting of `..` or start with a leading slash (`/`).

flags

Flags that affect the function's behavior:

- `NAME_FLAG_ATTACH_GLOBAL` — attach the name globally instead of locally. See the `gns` utility.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `name_attach()`, [name_close\(\)](#) (p. 1848), [name_detach\(\)](#) (p. 1850), and [name_open\(\)](#) (p. 1852) functions provide the basic pathname-to-server-connection mapping, without having to become a *full* resource manager.

If you've already created a dispatch structure, pass it in as the *dpp*. If you provide your own *dpp*, set *flags* to `NAME_FLAG_DETACH_SAVEDPP` when calling *name_detach()*; otherwise, your *dpp* is detached and destroyed automatically.

If you choose to pass a NULL as the *dpp*, *name_attach()* calls [dispatch_create\(\)](#) (p. 433) and [resmgr_attach\(\)](#) (p. 2580) internally to create a channel, however, it doesn't set any channel flag by itself. The created channel will have the `_NTO_CHF_DISCONNECT`, `_NTO_CHF_COID_DISCONNECT` and `_NTO_CHF_UNBLOCK` flags set.

The *name_attach()* function puts the name *path* into the path namespace under `/dev/name/[local|global]/path`. The name is attached locally by default, or globally when you set `NAME_FLAG_ATTACH_GLOBAL` in the *flags*. You can see attached names in `/dev/name/local` and `/dev/name/global` directories.



If the global name service, gns, is running on your system, the server application that calls *name_attach()* receives an `_IO_CONNECT` message with a subtype of `_IO_CONNECT_OPEN` when a client calls [name_open\(\)](#) (p. 1852). The server has to handle this message properly with a reply of `EOK` to allow *name_open()* to connect.

If the receive buffer that the server provides isn't large enough to hold a pulse, then [MsgReceive\(\)](#) (p. 1760) returns -1 with *errno* (p. 539) set to `EFAULT`.

name_attach_t

The *name_attach()* function returns a pointer to a `name_attach_t` structure that looks like this:

```
typedef struct _name_attach {
    dispatch_t* dpp;
    int         chid;
    int         mntid;
    int         zero[2];
} name_attach_t;
```

The members include:

dpp

The dispatch handle used in the creation of this connection.

chid

The channel ID used for *MsgReceive()* directly.

mntid

the mount ID for this name.

The information that's generally required by a server using these services is the *chid*.

The `gns` utility must be started and running first before an application can call `name_attach()` to globally advertise (or attach) a service (i.e. one represented by a path name).



If an application attaches a service locally, then applications from another machine can't lookup this service through the `gns` utility. If an application attaches its services globally, then any machine that's on the network and is running the `gns` manager can access the services.

An application can attach a service locally, only if there isn't another application that's attached locally to the same service. There's no credential restriction for applications that are attached as local services. An application can attach a service globally only if the application has `root` privilege.

Returns:

A pointer to a filled-in `name_attach_t` structure, or `NULL` if the call fails (*errno* (p. 539) is set).

Errors:

EBUSY

An error occurred when the function tried to create a channel; see *ChannelCreate()* (p. 278).

EEXIST

The specified path already exists.

EINVAL

An argument was invalid. For example, the *path* argument was `NULL`, the path was empty, it started with a leading slash (`/`), or it contained `.` components.

ENOMEM

There wasn't enough free memory to complete the operation.

ENOTDIR

A component of the pathname wasn't a directory entry.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/dispatch.h>
```



```

#define ATTACH_POINT "myname"

/* We specify the header as being at least a pulse */
typedef struct _pulse msg_header_t;

/* Our real data comes after the header */
typedef struct _my_data {
    msg_header_t hdr;
    int data;
} my_data_t;

/** Server Side of the code */
int server() {
    name_attach_t *attach;
    my_data_t msg;
    int rcvid;

    /* Create a local name (/dev/name/local/...) */
    if ((attach = name_attach(NULL, ATTACH_POINT, 0)) == NULL) {
        return EXIT_FAILURE;
    }

    /* Do your MsgReceive's here now with the chid */
    while (1) {
        rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);

        if (rcvid == -1) { /* Error condition, exit */
            break;
        }

        if (rcvid == 0) { /* Pulse received */
            switch (msg.hdr.code) {
                case _PULSE_CODE_DISCONNECT:
                    /*
                     * A client disconnected all its connections (called
                     * name_close() for each name_open() of our name) or
                     * terminated
                     */
                    ConnectDetach(msg.hdr.scoid);
                    break;
                case _PULSE_CODE_UNBLOCK:
                    /*
                     * REPLY blocked client wants to unblock (was hit by
                     * a signal or timed out). It's up to you if you
                     * reply now or later.
                     */
                    break;
                default:
                    /*
                     * A pulse sent by one of your processes or a
                     * _PULSE_CODE_COIDDEATH or _PULSE_CODE_THREADDEATH
                     * from the kernel?
                     */
                    break;
            }
            continue;
        }

        /* If the Global Name Service (gns) is running, name_open()
         sends a connect message. The server must EOK it. */
        if (msg.hdr.type == _IO_CONNECT) {
            MsgReply(rcvid, EOK, NULL, 0);
            continue;
        }

        /* Some other I/O message was received; reject it */
        if (msg.hdr.type > _IO_BASE && msg.hdr.type <= _IO_MAX) {
            MsgError(rcvid, ENOSYS);
            continue;
        }

        /* A message (presumably ours) received, handle */
        printf("Server receive %d \n", msg.data);
        MsgReply(rcvid, EOK, 0, 0);
    }

    /* Remove the name from the space */
    name_detach(attach, 0);
}

```

```

    }    return EXIT_SUCCESS;
}

/** Client Side of the code */
int client() {
    my_data_t msg;
    int server_coid;

    if ((server_coid = name_open(ATTACH_POINT, 0)) == -1) {
        return EXIT_FAILURE;
    }

    /* We would have pre-defined data to stuff here */
    msg.hdr.type = 0x00;
    msg.hdr.subtype = 0x00;

    /* Do whatever work you wanted with server connection */
    for (msg.data=0; msg.data < 5; msg.data++) {
        printf("Client sending %d \n", msg.data);
        if (MsgSend(server_coid, &msg, sizeof(msg), NULL, 0) == -1) {
            break;
        }
    }

    /* Close the connection */
    name_close(server_coid);
    return EXIT_SUCCESS;
}

int main(int argc, char **argv) {
    int ret;

    if (argc < 2) {
        printf("Usage %s -s | -c \n", argv[0]);
        ret = EXIT_FAILURE;
    }
    else if (strcmp(argv[1], "-c") == 0) {
        printf("Running Client ... \n");
        ret = client(); /* see name_open() for this code */
    }
    else if (strcmp(argv[1], "-s") == 0) {
        printf("Running Server ... \n");
        ret = server(); /* see name_attach() for this code */
    }
    else {
        printf("Usage %s -s | -c \n", argv[0]);
        ret = EXIT_FAILURE;
    }
    return ret;
}

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

As a server, you shouldn't assume that you're doing a *MsgReceive()* on a clean channel. Anyone can create a random message and send it to a process or a channel.

We recommend that you do the following to assure that you're playing safely with others in the system:

```
#include <sys/neutrino.h>

/* All of your messages should start with this header */
typedef struct _pulse msg_header_t;

/* Now your real data comes after this */
typedef struct _my_data {
    msg_header_t  hdr;
    int          data;
} my_data_t;
```

where:

hdr

Contains a *type/subtype* field as the first 4 bytes. This allows you to identify data which isn't destined for your server.

data

Specifies the receive data structure. The structure must be large enough to contain at least a pulse (which conveniently starts with the *type/subtype* field of most normal messages), because you'll receive a disconnect pulse when clients are detached.

name_close()

Close a server connection that was opened by [name_open\(\)](#)

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int name_close( int coid );
```

Arguments:

coid

A side-channel connection ID returned by [name_open\(\)](#) (p. 1852).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *name_close()* function closes a server connection that was opened by [name_open\(\)](#) (p. 1852).

Returns:

Zero for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

The side-channel connection ID specified by *coid* is invalid.

EINTR

The *name_close()* call was interrupted by a signal.

ENOSYS

The *name_close()* function isn't implemented for the filesystem specified by *coid*.

Examples:

See the “Client side of the code” section in [name_attach\(\)](#) (p. 1844).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

name_detach()

Remove a name from the namespace and destroy the channel

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int name_detach( name_attach_t * attach,
                 unsigned flags );
```

Arguments:

attach

A pointer to the `name_attach_t` structure returned by a successful call to [name_attach\(\)](#) (p. 1842).

flags

Flags that affect the function's behavior:

- `NAME_FLAG_DETACH_SAVEDPP` — don't destroy the dispatch handle.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `name_detach()` function removes the name from the namespace and destroys the channel created by [name_attach\(\)](#) (p. 1842). If you set `NAME_FLAG_DETACH_SAVEDPP` in `flags`, the dispatch pointer contained in the `name_attach_t` structure isn't destroyed; it's up to you to destroy it by calling [dispatch_destroy\(\)](#) (p. 438). The default is to destroy the dispatch pointer.

Returns:

Zero on success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

The mount ID (*mntid*) was never attached with [name_attach\(\)](#) (p. 1842).

Examples:

See [name_attach\(\)](#) (p. 1844) and [resmgr_detach\(\)](#) (p. 2602).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

name_open()

Open a name for a server connection

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int name_open( const char * name,
               int flags );
```

Arguments:

name

The name that you want to open for a server connection.

flags

Flags that affect the function's behavior:

- **NAME_FLAG_ATTACH_GLOBAL** — search for the name globally instead of locally.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *name_open()* function opens *name* for a server connection. No ordering is guaranteed when accessing resources on other nodes.



If the global name service, `gns`, is running on your system, an `_IO_CONNECT` message with a subtype of `_IO_CONNECT_OPEN` is sent to the server when a client calls *name_open()*. The server application must be able handle this message. For an example, see the documentation for [name_attach\(\)](#) (p. 1844).

Returns:

A nonnegative integer representing a side-channel connection ID (see [ConnectAttach\(\)](#) (p. 351)), or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied on a component of the *name*.

EBADFSYS

While attempting to open the named file, either the file itself or a component of the path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You'll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY

The connection specified by *name* has already been opened and additional connections aren't permitted.

EINTR

The *name_open()* operation was interrupted by a signal.

EISDIR

The named path is a directory.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENFILE

Too many files are currently open in the system.

ENOENT

The connection specified by *name* doesn't exist.

ENOTDIR

A component of the *name* prefix isn't a directory.

Examples:

See [name_attach\(\)](#) (p. 1844).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nanosleep()

Suspend a thread until a timeout or signal occurs

Synopsis:

```
#include <time.h>

int nanosleep( const struct timespec* rqtp,
               struct timespec* rmtp );
```

Arguments:

rqtp

A pointer to a [timespec](#) (p. 3473) structure that specifies the time interval for which you want to suspend the thread.

rmtp

NULL, or a pointer to a `timespec` structure where the function can store the amount of time remaining in the interval (the requested time minus the time actually slept).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `nanosleep()` function causes the calling thread to be suspended from execution until either:

- The time interval specified by the *rqtp* argument has elapsed
- Or
- A signal is delivered to the thread, and the signal's action is to invoke a signal-catching function or terminate the process.



This function can interfere with the kernel's efforts to manage power usage. The suspension time may be longer than requested because the argument value is rounded up to be a multiple of the system timer resolution (see the [Tick, Tock: Understanding the Microkernel's Concept of Time](#) chapter of the

QNX Neutrino *Programmer's Guide*) or because of scheduling and other system activity.

Returns:

0

The requested time has elapsed.

-1

The *nanosleep()* function was interrupted by a signal ([errno](#) (p. 539) is set).

Errors:**EAGAIN**

All timers are in use. You'll have to wait for a process to release one.

EFAULT

A fault occurred trying to access the buffers provided.

EINTR

The *nanosleep()* function was interrupted by a signal.

EINVAL

The number of nanoseconds specified by the *tv_nsec* member of the [timespec](#) (p. 3473) structure pointed to by *rqtp* is less than zero or greater than or equal to 1000 million.

Classification:

[POSIX 1003.1 TMR](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

nanospin()

Busy-wait without thread blocking for a period of time

Synopsis:

```
#include <time.h>

int nanospin( const struct timespec *when );
```

Arguments:

when

A pointer to a [timespec](#) (p. 3473) structure that specifies the amount of time to busy-wait for. This is a duration, not an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *nanospin()* function occupies the CPU for the amount of time specified by the argument *when* without blocking the calling thread. (The thread isn't taken off the ready list.) The function is essentially a `do...while` loop.

The *nanospin*()* functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX [timer_*](#) (p. 3427) functions.



Because of the nature of time measurement, the function might actually wait longer than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

The first time you call *nanospin()*, the C library invokes [nanospin_calibrate\(\)](#) (p. 1859) with an argument of 0 (interrupts enabled), if you haven't already called it.

Returns:

EOK

Success.

E2BIG

The delay specified by *when* is greater than 500 milliseconds.



A delay of more than a few milliseconds might not work on some processors. For longer delays, use the POSIX [timer_*\(\)](#) (p. 3427) functions.

EINTR

A too-high rate of interrupts occurred during the calibration routine.

ENOSYS

This system's *startup*-* program didn't initialize the timing information necessary to use *nanospin()*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

You should use busy-waiting only when absolutely necessary for accessing hardware. It isn't safe to call this function in an interrupt handler if *nanospin_calibrate()* hasn't been called yet.

nanospin_calibrate()

Calibrate before calling nanospin()*

Synopsis:

```
#include <time.h>

int nanospin_calibrate( int disable );
```

Arguments:

disable

1 to disable interrupts during the call to *nanospin_calibrate()*, or 0 to enable them; see below.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *nanospin_calibrate()* function performs the calibration for the *nanospin()** family of delay functions. The first time that you call [nanospin\(\)](#) (p. 1857), [nanospin_ns\(\)](#) (p. 1863), or [nanospin_ns_to_count\(\)](#) (p. 1865), the C library invokes *nanospin_calibrate()* with an argument of 0 (interrupts enabled), unless you call it directly first.

If you don't directly invoke *nanospin_calibrate()*, the first *nanospin*()* call in a process will have an overly long delay.



The *nanospin*()* functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX [timer_*\(\)](#) (p. 3427) functions.

Interrupts occurring during *nanospin_calibrate()* can throw off its timings. If *disable* is 0 (zero), you can prevent this situation by:

1. letting the thread acquire I/O privilege
2. disabling the interrupts around the *nanospin_calibrate()* call.

If *disable* is 1 (one), the code disables interrupts around the calibration loop(s). The calling thread is still responsible for obtaining I/O privilege before calling *nanospin_calibrate()*.

In order to make startup faster and reduce jitter, *nanospin_calibrate()* tries to read the calibration data from values stored in the system page. The startup for some boards includes an -o option that you can use to specify the calibration data (100 loop time and overhead) to store in the system page.

Returns:

EOK

Success.

EINTR

A too-high rate of interrupts occurred during the calibration routine.

EPERM

The process doesn't have superuser capabilities.

Examples:

Busy-wait for 100 nanoseconds:

```
#include <time.h>
#include <sys/syspage.h>

int disable = 0;
unsigned long time = 100;

...
/* Wake up the hardware, then wait for it to be ready. */
if ( (nanospin_calibrate( disable )) == EOK )
    nanospin_count( nanospin_ns_to_count( time ) );
else
    printf ("Didn't calibrate successfully.\n");
/* Use the hardware. */
...
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nanospin_count()

Busy-wait without blocking for a number of iterations

Synopsis:

```
#include <time.h>

void nanospin_count( unsigned long count );
```

Arguments:

count

The number of iterations that you want to busy-wait for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *nanospin_count()* function busy-waits for the number of iterations specified in *count*. Use [nanospin_ns_to_count\(\)](#) (p. 1865) to turn a number of nanoseconds into an iteration count suitable for *nanospin_count()*.



The *nanospin*()* functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX [timer_*\(\)](#) (p. 3427) functions.

Examples:

Busy-wait for at least 100 nanoseconds:

```
#include <time.h>
#include <sys/syspage.h>

unsigned long time = 100;

...
/* Wake up the hardware, then wait for it to be ready. */
nanospin_count( nanospin_ns_to_count( time ) );
/* Use the hardware. */
...
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

You should use busy-waiting only when absolutely necessary for accessing hardware.

nanospin_ns()

Busy-wait without blocking for a period of time

Synopsis:

```
#include <time.h>

int nanospin_ns( unsigned long nsec );
```

Arguments:

nsec

The number of nanoseconds that you want to busy-wait for.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *nanospin_ns()* function busy-waits for the number of nanoseconds specified in *nsec*, without blocking the calling thread.

The *nanospin*()* functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX [timer_*\(\)](#) (p. 3427) functions.



Because of the nature of time measurement, the function might actually wait longer than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

The first time you call *nanospin_ns()*, the C library invokes [nanospin_calibrate\(\)](#) (p. 1859) with an argument of 0 (interrupts enabled), if you haven't invoked it directly first.

Returns:

EOK

Success.

E2BIG

The delay specified by *nsec* is greater than 500 milliseconds.



A delay of more than a few milliseconds might not work on some processors. For longer delays, use the POSIX [timer_*\(\)](#) (p. 3427) functions.

EINTR

A too-high rate of interrupts occurred during the calibration routine.

ENOSYS

This system's `startup-*` program didn't initialize the timing information necessary to use `nanospin_ns()`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

You should use busy-waiting only when absolutely necessary for accessing hardware. It isn't safe to call this function in an interrupt handler if `nanospin_calibrate()` hasn't been called yet.

nanospin_ns_to_count()

Convert a time in nanoseconds into a number of iterations

Synopsis:

```
#include <time.h>

unsigned long nanospin_ns_to_count(
    unsigned long nsec );
```

Arguments:

nsec

The number of nanoseconds that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *nanospin_ns_to_count()* function converts the number of nanoseconds specified in *nsec* into an iteration count suitable for [nanospin_count\(\)](#) (p. 1861).



The *nanospin*()* functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX [timer_*\(\)](#) (p. 3427) functions.

The first time that you call *nanospin_ns_to_count()*, the C library invokes [nanospin_calibrate\(\)](#) (p. 1859) with an argument of 0 (interrupts enabled), if you haven't invoked it directly first.

Returns:

The amount of time to busy-wait, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINTR

A too-high rate of interrupts occurred during the calibration routine.

ENOSYS

This system's startup- * program didn't initialize the timing information necessary to use *nanospin_ns_to_count()*.

Examples:

Busy-wait for at least one nanosecond:

```
#include <time.h>
#include <sys/syspage.h>

unsigned long time = 1;

...
/* Wake up the hardware, then wait for it to be ready. */

/*
The C library invokes nanospin_calibrate
if it hasn't already been called.
*/

nanospin_count( nanospin_ns_to_count( time ) );

/* Use the hardware. */
...
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

You should use busy-waiting only when absolutely necessary for accessing hardware. It isn't safe to call this function in an interrupt handler if *nanospin_calibrate()* hasn't been called yet.

nap()

Sleep for a given number of milliseconds

Synopsis:

```
#include <unix.h>

unsigned int nap( unsigned int ms);
```

Arguments:

ms

The number of milliseconds that you want the process to sleep.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `nap()` routine delays the calling process for *ms* milliseconds. This function is the same as [delay\(\)](#) (p. 401) and is similar to [napms\(\)](#) (p. 1868).



The suspension time may be greater than the requested amount, due to the nature of time measurement (see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*), or due to the scheduling of other, higher-priority threads by the system.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

napms()

Sleep for a given number of milliseconds

Synopsis:

```
#include <curses.h>

int napms( int ms);
```

Arguments:

ms

The number of milliseconds that you want the process to sleep.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *napms()* routine delays the calling process for *ms* milliseconds. This function is similar to [delay\(\)](#) (p. 401) and [nap\(\)](#) (p. 1867).



The suspension time may be greater than the requested amount, due to the nature of time measurement (see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*), or due to the scheduling of other, higher-priority threads by the system.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nbaconnect()

Initiate a connection on a socket (nonblocking)

Synopsis:

```
#include <sys/socket.h>

int nbaconnect( int s,
                const struct sockaddr * name,
                size_t namelen );
```

Arguments:

s

The descriptor of the socket on which to initiate the connection.

name

The name of the socket to connect to for a SOCK_STREAM connection.

namelen

The length of the *name*, in bytes.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *nbaconnect()* function is called in place of [connect\(\)](#) (p. 348), to prevent a nonblocking *connect()* from blocking during an autoconnect (see /etc/autoconnect).

When the autoconnect behavior is used, *connect()* may block your application while waiting for the autoconnect to complete; *nbaconnect()* allows your application to continue executing during the autoconnect.

The *nbaconnect()* function takes the same arguments as [connect\(\)](#) (p. 348), but it differs in the return value when an autoconnect is required. If an autoconnect is required, a file descriptor (*fd*) is returned. The *fd* is used in the call to [nbaconnect_result\(\)](#) (p. 1871) to get the *errno* related to the autoconnect and the connect attempt.



Since *nbaconnect_result()* is a blocking call, it's recommended that you call *select()* (p. 2791) first to determine if there's data available on the pipe.

When the data's available, call *nbaconnect_result()* (p. 1871) to get the status of the *nbaconnect()* attempt.

If an autoconnect isn't required, *nbaconnect()* returns -1 and exhibits the same behavior as *connect()* on nonblocking sockets (e.g. if -1 is returned and *errno* is set to *EINPROGRESS*, it's possible to do a *select()* for completion by selecting the socket for writing).

Returns:

A file descriptor that you can pass to *nbaconnect_result()* to get the result of the *nbaconnect()* attempt, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

Any value from the Errors section in *connect()* (p. 349), as well as:

EINVAL

The socket file descriptor being passed isn't nonblocking.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The pipe manager must be available.

nbaconnect_result()

Get the status of the previous call to nbaconnect()

Synopsis:

```
#include <sys/socket.h>

int nbaconnect_result( int fd,
                     int * error );
```

Arguments:

fd

The file descriptor returned by [nbaconnect\(\)](#) (p. 1869).

error

A pointer to a location where the function can store the status.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *nbaconnect_result()* function gets the status of the previous [nbaconnect\(\)](#) (p. 1869) call when an *fd* was returned. Since *nbaconnect_result()* is a blocking call, it's best to test the status of the *fd* with a call to [select\(\)](#) (p. 2791) to verify that the file descriptor is ready to be read.

When there's data available, the status is put in *error*, which may be any of the [errno](#) (p. 539) values set by [connect\(\)](#) (p. 348) during an attempt to make a non-blocking connection.

The *fd* is always closed by this function whether or not there's a status to report.

Returns:

0

The call was successful; *error* contains the status.

-1

An error occurred while obtaining the status.

Errors:

Any value from the Errors section in [connect\(\)](#) (p. 349), as well as:

EBADF

Invalid *fd*.

ENOMSG

There's no data, or not enough data, from the *fd*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ncurses

Control a terminal's display

Name:

`libncurses.a`, `libncursesS.a`

Description:

The `ncurses` library provides a set of functions that you can use to control a terminal's display. Use the `-l ncurses` option to `qcc` to link against this library.

For more information, see `<ncurses.h>`, as well as *Programming with curses* by John Strang, published by O'Reilly Media.

ND_NODE_CMP()

Compare two node descriptors

Synopsis:

```
#include <sys/netmgr.h>
#define ND_NODE_CMP(a, b) ...
```

Arguments:

a, b

The node descriptors that you want to compare. You can use either the value 0 or ND_LOCAL_NODE to refer to the local node.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *ND_NODE_CMP()* macro compares two node descriptors.

Returns:

< 0

The node descriptor *a* is less than *b*.

0

The descriptors refer to the same machine.

> 0

The node descriptor *a* is greater than *b*.

Examples:

```
#include <sys/neutrino.h>
uint32_t nd1, nd2;
if ( ND_NODE_CMP(nd1, nd2) == 0 ) {
    /* Same node */
    ...
}
```

```
} else {  
    /* Different nodes */  
    ...  
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

netent

Structure for information from the network database

Synopsis:

```
#include <netdb.h>

struct netent {
    char *   n_name;
    char **  n_aliases;
    int      n_addrtype;
    uint32_t n_net;
};
```

Description:

This structure holds information from the network database, `/etc/networks`. The members of this structure are:

n_name

The name of the network.

n_aliases

A zero-terminated list of alternate names for the network.

n_addrtype

The type of the network number returned; currently only `AF_INET`.

n_net

The network number. Network numbers are returned in machine-byte order.

Classification:

[*POSIX 1003.1*](#) (p. 46)

netmgr_ndtostr()

Convert a node descriptor into a string

Synopsis:

```
#include <sys/netmgr.h>

int netmgr_ndtostr( unsigned flags,
                    int nd,
                    char * buf,
                    size_t maxbuf );
```

Arguments:

flags

Which part(s) of the Fully Qualified Path Name (FQPN) to adjust; see below.

nd

The node descriptor that you want to convert.

buf

A pointer to a buffer where the function can store the converted identifier.

maxbuf

The size of the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *netmgr_ndtostr()* function converts a node descriptor, *nd*, to a string and stores the string in the buffer pointed to by *buf*. The size of the buffer is given by *maxbuf*.

A node descriptor is a temporary numeric description of a remote node. For more information, see the Qnet Networking chapter of the *System Architecture* guide.

The *flags* argument indicates which part(s) of the Fully Qualified Path Name (FQPN) to adjust. The following diagram shows the components for the FQPN, `/net/qnet.qnx.com~preferred`:

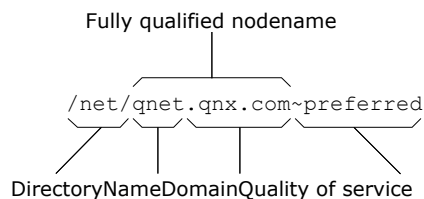


Figure 7: Components of a fully qualified pathname.

The Fully Qualified Node Name (FQNN) is `qnet.qnx.com`.

The default string that `netmgr_ndtostr()` builds is the FQNN, plus the Quality of Service (QoS) if it isn't the default (`~loadbalance`). You can pass this string to any other node that can call [netmgr_strtond\(\)](#) (p. 1883) and has a `nd` that refers to the same node with the same QoS.

A bitwise OR of *flags* modify the default string in the following way:

ND2S_DIR_HIDE

Never include the directory.

ND2S_DIR_SHOW

Always build a pathname to the root directory (i.e. `/`) of the node indicated by *nd*. For example, calling:

```
netmgr_ndtostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)
```

on a node with a default domain of `qnx.com` using a *nd* that refers to a FQNN of `peterv.qnx.com` results in the string, `/net/peterv.qnx.com/`.

If Qnet isn't active on the node, and `netmgr_ndtostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)` has a *nd* of `ND_LOCAL_NODE`, then the string is `/`.

ND2S_DOMAIN_HIDE

Never include the domain.

ND2S_DOMAIN_SHOW

Always include the domain.

ND2S_LOCAL_STR

Display shorter node names on your local node. For example, calling:

```
netmgr_ndtostr(ND2S_LOCAL_STR, nd, buf, sizeofbuf)
```

on a node with a default domain of `qnx.com` using a *nd* that refers to an FQPN of `/net/peterv.qnx.com` results in a string of `peterv`.

Whereas a *nd* that refers to `/net/peterv.anotherhost.com` results in `peterv.anotherhost.com`.

ND2S_NAME_HIDE

Never include the name.

ND2S_NAME_SHOW

Always include the name.

ND2S_QOS_HIDE

Never include the quality of service (QoS).

ND2S_QOS_SHOW

Always include the QoS.

ND2S_SEP_FORCE

Always include a leading separator. For example, calling:

```
netmgr_ndtostr(ND2S_SEP_FORCE | ND2S_DIR_HIDE |
               ND2S_NAME_HIDE | ND2S_DOMAIN_HIDE |
               ND2S_QOS_SHOW, nd, buf, sizeofbuf)
```

with a *nd* of `ND_LOCAL_NODE` results in a string of `~loadbalance`. This is useful if you want to concatenate each component of the FQPN individually.



Don't use a `ND2S_*_HIDE` and a corresponding `ND2S_*_SHOW` together.

Returns:

The length of the string, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**ENOTSUP**

Qnet isn't running.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/netmgr.h>

int main ()
{
    int nd1, nd2, nd3, len;
    char path1[50] = "/net/dave",
```

```

        path2[50] = "/net/karen",
        buff[100];

    nd1 = netmgr_strtond( path1, NULL);
    if (nd1 == -1) {
        perror ("netmgr_strtond" );
        return EXIT_FAILURE;
    }
    else {
        printf ("Node id for %s is %d.\n", path1, nd1);
    }

    nd2 = netmgr_strtond( path2, NULL);
    if (nd2 == -1) {
        perror ("netmgr_strtond" );
        return EXIT_FAILURE;
    }
    else {
        printf ("Node id for %s is %d.\n", path2, nd2);
    }

    nd3 = netmgr_remote_nd ( nd2, nd1 );
    if (nd3 == -1) {
        perror ("netmgr_strtond" );
        return EXIT_FAILURE;
    }
    else {
        printf ("Node id for %s, relative to %s, is %d.\n",
            path1, path2, nd3);
    }

    len = netmgr_ndtostr ( ND2S_DIR_HIDE | ND2S_DOMAIN_SHOW |
        ND2S_NAME_SHOW | ND2S_QOS_SHOW, nd1, buff, 100 );
    if (len == -1) {
        perror ("netmgr_ndtostr" );
    }
    else {
        printf ("Node name for %d is %s.\n", nd1, buff);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

netmgr_remote_nd()

Get a node descriptor that's relative to a remote node

Synopsis:

```
#include <sys/netmgr.h>

int netmgr_remote_nd( int remote_nd,
                    int local_nd );
```

Arguments:

remote_nd

The node descriptor of a remote node.

local_nd

A node descriptor, relative to the local node, that you want to convert to be relative to the remote node.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *netmgr_remote_nd()* function converts a node descriptor that's relative to the local node into a node descriptor that's relative to the specified remote node.

Returns:

The node descriptor, relative to the remote node, or `-1` if an error occurred (*errno* (p. 539) is set).

Examples:

See [netmgr_ndtostr\(\)](#) (p. 1877).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

netmgr_strtond()

Convert a string into a node descriptor

Synopsis:

```
#include <sys/netmgr.h>

int netmgr_strtond( const char * nodename,
                   char ** endstr );
```

Arguments:

nodename

The string that you want to convert into a node descriptor.

endstr

NULL, or the address of a location where the function can store a pointer to the character after the node name in the string.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *netmgr_strtond()* function converts a string to a node descriptor. If *endstr* isn't NULL, it's set to point to the character after the node name in the given string.

Returns:

The node descriptor, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

ENOTSUP

Qnet isn't running.

Examples:

See [netmgr_ndtostr\(\)](#) (p. 1877).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nextafter(), nextafterf()

Compute the next representable double-precision floating-point number

Synopsis:

```
#include <math.h>

double nextafter ( double x,
                  double y);

float nextafterf ( float x,
                  float y );
```

Arguments:

- x***
The number that you want the next number after.
- y***
A number that specifies the direction you want to go; see below.

Library:

libm
Use the -l m option to qcc to link against this library.

Description:

The *nextafter()* and *nextafterf()* functions compute the next representable double-precision floating-point value following *x* in the direction of *y*.

Returns:

The next machine floating-point number of *x* in the direction towards *y*.

If:	<i>nextafter()</i> returns:
<i>y</i> < <i>x</i>	The next possible floating-point value less than <i>y</i>
<i>y</i> > <i>x</i>	The next possible floating-point value greater than <i>x</i>
<i>x</i> is NAN	NAN
<i>y</i> is NAN	NAN

If:	<i>nextafter()</i> returns:
x is finite	$\pm$ HUGE_VAL, according to the sign of x (<i>errno</i> is set)



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set *errno* (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

void dump_to_hex(double d) {
    printf("0x%08x %08x \n",
        (uint32_t)((uint64_t*)&d) >> 32),
        (uint32_t)((uint64_t*)&d));
}

int main(int argc, char** argv)
{
    double a, b, c;

    a = 0 ;
    b = nextafter(a, -1);
    c = nextafter(a, 1);
    printf("Next possible value before %f is %f \n", a, b);
    printf("-->"); dump_to_hex(a);
    printf("-->"); dump_to_hex(b);
    printf("Next possible value after %f is %f \n", a, c);
    printf("-->"); dump_to_hex(a);
    printf("-->"); dump_to_hex(c);

    return(0);
}
```

produces the output:

```
Next possible value before 0.000000 is 0.000000
-->0x00000000 00000000
-->0x80000000 00000001
Next possible value after 0.000000 is 0.000000
-->0x00000000 00000000
```

Classification:

ANSI (p. 46), *POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

nftw()*, *nftw64()

Walk a file tree

Synopsis:

```
#include <ftw.h>

int nftw( const char *path,
          int (*fn)( const char *fname,
                    const struct stat *sbuf,
                    int flags,
                    struct FTW *ftw),
          int depth,
          int flags);
```

Arguments:

path

The path of the directory whose file tree you want to walk.

fn

A pointer to a function that you want to call for each file; see below.

depth

The maximum number of file descriptors that *nftw()* can use. The *nftw()* function uses one file descriptor for each level in the tree.

If *depth* is zero or negative, the effect is the same as if it were 1. The *depth* must not be greater than the number of file descriptors currently available for use. The *nftw()* function is faster if *depth* is at least as large as the number of levels in the tree.

flags

The value of *flags* is constructed by the bitwise ORing of values from the following list, defined in the `<ftw.h>` header file.

FTW_CHDIR

If set, *nftw()* changes the current working directory to each directory as it reports files in that directory.

FTW_DEPTH

If set, *nftw()* reports all files in a directory before reporting the directory itself (otherwise the directory is reported before any file it contains).

FTW_MOUNT

If set, *nftw()* only reports files on the same filesystem as *path*.

FTW_PHYS

If set, *nftw()* performs a physical walk and doesn't follow any symbolic link.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



These functions are in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *nftw()* and *nftw64()* functions recursively descend the directory hierarchy identified by *path*. The *nftw64()* function is a 64-bit version of *nftw()*. For each object in the hierarchy, *nftw()* calls the user-defined function *fn()*, passing to it:

- a pointer to a NULL-terminated character string containing the name of the object
- a pointer to a `stat` structure (see [stat\(\)](#) (p. 3164)) containing information about the object
- an integer. Possible values of the integer, defined in the `<ftw.h>` header, are:

FTW_F

The object is a file.

FTW_D

The object is a directory.

FTW_DNR

The object is a directory that can't be read. Descendents of the directory aren't processed.

FTW_DP

The object is a directory, and its contents have been reported. See the `FTW_DEPTH` flag above.

FTW_NS

The `stat()` failed on the object because the permissions weren't appropriate. The `stat` buffer passed to `fn()` is undefined.

FTW_SL

The object is a symbolic link. See the `FTW_PHYS` flag above.

FTW_SLN

The object is a symbolic link that does not name an existing file.

- a pointer to a `FTW` structure, which contains the following fields:

base

The offset of the object's filename in the pathname passed as the first argument to `fn()`.

level

The depth relative to the root of the walk (where the root is level 0).

quit

A flag that can be set to control the behaviour of `nftw()` within the current directory. If assigned, it may be given the following values:

FTW_SKR

Skip the remainder of this directory

FTW_SKD

If the object is `FTW_D`, then do not enter into this directory.

The tree traversal continues until the tree is exhausted, an invocation of `fn()` returns a nonzero value, or some error is detected within `nftw()` (such as an I/O error). If the tree is exhausted, `nftw()` returns zero. If `fn()` returns a nonzero value, `nftw()` stops its tree traversal and returns whatever value was returned by `fn()`.

When `nftw()` returns, it closes any file descriptors it opened; it doesn't close any file descriptors that may have been opened by `fn()`.

Returns:

0

Success.

-1

An error (other than EACCES) occurred (*errno* (p. 539) is set).

Classification:

nftw() is [POSIX 1003.1 XSI](#) (p. 46); *nftw64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Because *nftw()* is recursive, it might terminate with a memory fault when applied to very deep file structures.

This function uses *malloc()* to allocate dynamic storage during its operation. If *nftw()* is forcibly terminated, for example if *longjmp()* is executed by *fn()* or an interrupt routine, *nftw()* doesn't have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn()* return a nonzero value at its next invocation.

nice()

Change the priority of a process

Synopsis:

```
#include <unistd.h>

int nice( int incr );
```

Arguments:

incr

The amount that you want to add to the process's priority.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *nice()* function allows a process to change its priority. The invoking process must be in a scheduling class that supports the operation.



This function has no effect in any of the scheduling policies that QNX Neutrino supports. Use [sched_setparam\(\)](#) (p. 2738), [pthread_setschedparam\(\)](#) (p. 2400), or [pthread_setschedprio\(\)](#) (p. 2402) instead of this function.

The *nice()* function adds the value of *incr* to the nice value of the calling process. A process's nice value is a nonnegative number; a greater positive value results in a lower CPU priority.

A maximum nice value of $2 * \text{NZERO} - 1$ and a minimum nice value of 0 are imposed by the system. `NZERO` is defined in `<limits.h>` with a default value of 20. If you request a value above or below these limits, the nice value is set to the corresponding limit. A nice value of 40 is treated as 39. Only a process with superuser privileges can lower the nice value.

Returns:

The new nice value minus `NZERO`. If an error occurred, -1 is returned, the process's nice value isn't changed, and [errno](#) (p. 539) is set.

Errors:

EINVAL

The *nice()* function was called by a process in a scheduling class other than time-sharing.

EPERM

The *incr* argument was negative or greater than 40, and the effective user ID of the calling process isn't the superuser.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *nice()*, and if it returns -1, check to see if *errno* is nonzero.

rand48()

Generate a pseudo-random nonnegative long integer in a thread-safe manner

Synopsis:

```
#include <stdlib.h>

long rand48( unsigned short xsubi[3] );
```

Arguments:

xsubi

An array that comprises the 48 bits of the initial value that you want to use.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *rand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative long integer uniformly distributed over the interval [0, 2³¹].

The *xsubi* array should contain the desired initial value; this makes *rand48()* thread-safe, and lets you start a sequence of random numbers at any known value.

Returns:

A pseudo-random long integer.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

nsec2timespec()

Convert nanoseconds to a timespec structure

Synopsis:

```
#include <time.h>

void nsec2timespec( struct timespec *timespec_p,
                   uint64_t nsec );
```

Arguments:

timespec_p

A pointer to a [timespec](#) (p. 3473) structure where the function can store the converted time.

nsec

The number of nanoseconds that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This function converts the given number of nanoseconds, *nsec*, into seconds and nanoseconds, and stores them in the `timespec` structure pointed to by *timespec_p*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

ntohl()

Convert a 32-bit value from network-byte order to host-byte order

Synopsis:

```
#include <arpa/inet.h>

uint32_t ntohl( uint32_t netlong );
```

Arguments:

netlong

The value that you want to convert.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *ntohl()* function converts a 32-bit value from network-byte order to host-byte order. If a machine's byte order is the same as the network order, this routine is defined as a null macro.

You most often use this routine in conjunction with internet addresses and ports returned by [gethostbyname\(\)](#) (p. 872) and [getservent\(\)](#) (p. 956).

Returns:

The value in host-byte order.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

ntohs()

Convert a 16-bit value from network-byte order to host-byte order

Synopsis:

```
#include <arpa/inet.h>

uint16_t ntohs( uint16_t netshort );
```

Arguments:

netshort

The value that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ntohs()* function converts a 16-bit value from network-byte order to host-byte order. If a machine's byte order is the same as the network order, this routine is defined as a null macro.

You most often use this routine in conjunction with internet addresses and ports returned by [gethostbyname\(\)](#) (p. 872) and [getservent\(\)](#) (p. 956).

Returns:

The value in host-byte order.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Chapter 17

O

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “O”.

offsetof()

Return the offset of an element within a structure

Synopsis:

```
#include <stddef.h>

#define offsetof( composite, name ) ...
```

Arguments:

composite

A struct or union.

name

The name of an element in *composite*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *offsetof()* macro returns the offset of the element *name* within the struct or union *composite*.

This provides a portable method to determine the offset.

Returns:

The offset of *name*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

struct new_def
{
    char *first;
    char second[10];
    int third;
};

int main( void )
{
```



```
    printf( "first:%d second:%d third:%d\n",
            offsetof( struct new_def, first ),
            offsetof( struct new_def, second ),
            offsetof( struct new_def, third ) );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

This is a macro.

OMAPI

Object Management Application Programming Interface

Description:

OMAPI is an programming layer designed for controlling remote applications, and for querying them for their state. It is currently used by the ISC DHCP server and this outline addresses the parts of OMAPI appropriate to the clients of DHCP server. It does this by also describing the use of a thin API layered on top of OMAPI called `dhcpcctl`.

OMAPI uses TCP/IP as the transport for server communication, and security can be imposed by having the client and server cryptographically sign messages using a shared secret.

The `dhcpcctl*()` (p. 412) API works by presenting the client with handles to objects that act as surrogates for the real objects in the server. For example, a client will create a handle for a lease object, and will request the server to fill the lease handle's state. The client application can then pull details such as the lease expiration time from the lease handle.

Modifications can be made to the server state by creating handles to new objects, or by modifying attributes of handles to existing objects, and then instructing the server to update itself according to the changes made.

Usage:

The client application must always call `dhcpcctl_initialize()` before making calls to any other `dhcpcctl*()` functions. This initializes various internal data structures.

To create the connection to the server, the client must use `dhcpcctl_connect()`. As well as making the physical connection, it will also set up the connection data structures to do authentication on each message, if that is required.

All the `dhcpcctl*()` functions return an integer value of type `isc_result_t`. A successful call will yield a result of `ISC_R_SUCCESS`. If the call fails for a reason local to the client (e.g. insufficient local memory, or invalid arguments to the call) then the return value of the `dhcpcctl*()` function will show that. If the call succeeds but the server couldn't process the request the error value from the server is returned through another way, shown below.

The easiest way to understand `dhcpcctl*()` is to see it in action. The following program is fully functional, but almost all error checking has been removed to make is shorter and easier to understand. This program will query the server running on the localhost

for the details of the lease for IP address 10.0.0.101. It will then print out the time the lease ends.

```
#include <stdarg.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>

#include <isc/result.h>
#include <dhcpcctl/dhcpcctl.h>

int main (int argc, char **argv) {

    /* All modifications of handles and all accesses of
       handle data happen via dhcpcctl_data_string objects. */

    dhcpcctl_data_string ipaddrstring = NULL;
    dhcpcctl_data_string value = NULL;

    dhcpcctl_handle connection = NULL;
    dhcpcctl_handle lease = NULL;
    isc_result_t waitstatus;
    struct in_addr convaddr;
    time_t thetime;

    /* Required first step. */

    dhcpcctl_initialize ();

    /* Set up the connection to the server. The server
       normally listens on port 7911 unless configured to do
       otherwise. */

    dhcpcctl_connect (&connection, "127.0.0.1", 7911, 0);

    /* Create a handle to a lease. This call just
       sets up local data structure. The server hasn't
       yet made any association between the client's data
       structure and any lease it has. */

    dhcpcctl_new_object (&lease, connection, "lease");

    /* Create a new data string to store in the handle. */

    memset (&ipaddrstring, 0, sizeof ipaddrstring);
    inet_pton(AF_INET, "10.0.0.101", &convaddr);
    omapi_data_string_new (&ipaddrstring, 4, MDL);

    /* Set the ip-address attribute of the lease handle
       to the given address. We haven't set any
       other attributes, so when the server makes the
       association, the IP address will be all it uses to
       look up the lease in its tables. */

    memcpy(ipaddrstring->value, &convaddr.s_addr, 4);
    dhcpcctl_set_value (lease, ipaddrstring, "ip-address");

    /* Prime the connection with the request to look up
       the lease in the server and fill up the local handle
       with the attributes the server will send over in
       its answer. */

    dhcpcctl_open_object (lease, connection, 0);

    /* Send the message (to look up the lease and send back
       the attribute values in the answer) to the server.
       The value in the variable waitstatus when the function
       returns will be the result from the server. If the
       message couldn't be processed properly by the server,
       then the error will be reflected here. */
```

```

    dhcpctl_wait_for_completion (lease, &waitstatus);
    if (waitstatus != ISC_R_SUCCESS) {
        /* server not authoritative */
        exit (0);
    }

    /* Clean up memory we no longer need. */
    dhcpctl_data_string_dereference( &ipaddrstring, MDL);

    /* Get the attribute named "ends" from the lease handle.
       This is a 4-byte integer of the time (in Unix epoch
       seconds) that the lease will expire. */
    dhcpctl_get_value (&value, lease, "ends");
    memcpy(&thetime, value->value, value->len);
    dhcpctl_data_string_dereference(&value, MDL);

    fprintf (stdout, "ending time is %s",
            ctime(&thetime));
}

```

Authentication

If the server demands authenticated connections, then before opening the connection the user must call *dhcpctl_new_authenticator()*:

```

dhcpctl_handle authenticator = NULL;
const char *keyname = "a-key-name";
const char *algorithm = "hmac-md5";
const char *secret = "a-shared-secret";

dhcpctl_new_authenticator (&authenticator, keyname,
                          algorithm, secret,
                          strlen(secret) + 1);

```

The *keyname*, *algorithm*, and *secret* must all match what is specified in the server's `dhcpd.conf` file, excepting that the *secret* should appear in “raw” form, not in base-64 as it would in `dhcpd.conf`:

```

key "a-key-name" {
    algorithm hmac-md5;
    secret "a-shared-secret";
};

# Set the omapi-key value to use
# authenticated connections

omapi-key a-key-name;

```

The authenticator handle that is created by the call to *dhcpctl_new_authenticator()* must be given as the last (the fourth) argument to *dhcpctl_connect()*. All messages will then be signed with the given secret string using the specified algorithm.

Contributing author:

OMAPI was created by Ted Lemon of Nominum, Inc. Information about Nominum and support contracts for DHCP and BIND can be found at <http://www.nominum.com> This documentation was written by James Brister of Nominum, Inc.

open()*, *open64()

Open a file

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open( const char * path,
          int oflag,
          ... );

int open64( const char * path,
            int oflag,
            ... );
```

Arguments:

path

The path name of the file that you want to open.

oflag

Flags that specify the status and access modes of the file; see below.

If you set `O_CREAT` in *oflag*, you must also specify the following argument:

mode

An object of type `mode_t` that specifies the access mode that you want to use for a newly created file. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for *stat()*, and the description of `O_CREAT` (p. ?), below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *open()* and *open64()* functions open the file named by *path*, creating an open file description that refers to the file, and a file descriptor that refers to the file description.

The *open64()* function is a 64-bit version of *open()*. The file status flags and the file access modes of the open file description are set according to the value of *oflag*.



These functions ignore any advisory locks that you set with *fcntl()* (p. 623).

The open file descriptor created is new, and therefore isn't shared with any other process in the system.

Construct the value of *oflag* by bitwise ORing values from the following list, defined in the `<fcntl.h>` header file. You must specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY

Open for reading only.

O_RDWR

Open for reading and writing. Opening a FIFO for read-write is unsupported.

O_WRONLY

Open for writing only.

You can also specify any combination of the remaining flags in the value of *oflag*:

O_APPEND

If set, the file offset is set to the end of the file prior to each write.

O_ASYNC

(QNX Neutrino extension) Not currently supported.

O_CLOEXEC

(QNX Neutrino extension) Close the file descriptor on execution.

O_CREAT

This option requires a third argument, *mode*, which is of type `mode_t`. If the file exists, this flag has no effect, except in combination with **O_EXCL** as noted below.

Otherwise, the file is created; the file's user ID is set to the effective user ID of the process; the group ID is set to the effective group ID of the process or the group ID of the file's parent directory (see *chmod()* (p. 289)).

The permission bits, as defined in `<sys/stat.h>`, are set to the value of *mode*, except those bits set in the process's file mode creation mask (see *umask()* (p. 3557) for details). Bits set in *mode* other than the file permission

bits (i.e. the file type bits) are ignored. The *mode* argument doesn't affect whether the file is opened for reading, for writing, or for both.



The *open()* and *open64()* functions are varargs functions. If you specify `O_CREAT` but you don't provide a *mode* argument, they use whatever is on the stack as the *mode*.

O_DSYNC

If set, this flag affects subsequent I/O calls; each call to *write()* (p. 3737) waits until all data is successfully transferred to the storage device such that it's readable on any subsequent open of the file (even one that follows a system failure) in the absence of a failure of the physical storage medium. If the physical storage medium implements a non-write-through cache, then a system failure may be interpreted as a failure of the physical storage medium, and data may not be readable even if this flag is set and the *write()* indicates that it succeeded.

O_EXCL

If you set both `O_EXCL` and `O_CREAT`, *open()* fails if the file exists. The check for the existence of the file and the creation of the file if it doesn't exist are atomic; no other process that's attempting the same operation with the same filename at the same time will succeed. Specifying `O_EXCL` without `O_CREAT` has no effect.

O_LARGEFILE

(QNX Neutrino extension) Allow the file offset to be 64 bits long.

O_NOCTTY

If set, and *path* identifies a terminal device, the *open()* function doesn't cause the terminal device to become the controlling terminal for the process.

O_NOFOLLOW

(QNX Neutrino extension) If the last component in the path is a symlink, make the open fail with an error of `ELOOP`.

O_NONBLOCK

- When opening a FIFO with `O_RDONLY` or `O_WRONLY` set:

If `O_NONBLOCK` is set:

Calling *open()* for reading-only returns without delay. Calling *open()* for writing-only returns an error if no process currently has the FIFO open for reading.

If O_NONBLOCK is clear:

Calling *open()* for reading-only blocks until a process opens the file for writing. Calling *open()* for writing-only blocks until a process opens the file for reading.

- When opening a block special or character special file that supports nonblocking opens:

If O_NONBLOCK is set:

The *open()* function returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

If O_NONBLOCK is clear:

The *open()* function waits until the device is ready or available before returning. The definition of when a device is ready is device-specific.

- Otherwise, the behavior of O_NONBLOCK is unspecified.

O_NOSYMLINK

(QNX Neutrino extension) If any component in the path is a symlink, make the open fail with an error of ELOOP.

O_REALIDS

(QNX Neutrino extension) Use the real uid/gid for permissions checking.

O_RSYNC

Read I/O operations on the file descriptor complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags.

O_SYNC

If set, this flag affects subsequent I/O calls; each call to [read\(\)](#) (p. 2502) or [write\(\)](#) (p. 3737) is complete only when both the data has been successfully transferred (either read or written) and all file system information relevant to that I/O operation (including that required to retrieve said data) is successfully transferred, including file update and/or access times, and so on. See the discussion of a successful data transfer in O_DSYNC, above.

O_TRUNC

If the file exists and is a regular file, and the file is successfully opened `O_WRONLY` or `O_RDWR`, the file length is truncated to zero and the mode and owner are left unchanged. `O_TRUNC` has no effect on FIFO or block or character special files or directories. Using `O_TRUNC` with `O_RDONLY` has no effect.

The largest value that can be represented correctly in an object of type `off_t` shall be established as the offset maximum in the open file description.

Returns:

A nonnegative integer representing the lowest numbered unused file descriptor. On a file capable of seeking, the file offset is set to the beginning of the file. Otherwise, -1 is returned ([errno](#) (p. 539) is set).



In QNX Neutrino, the returned file descriptor is the same as the connection ID (or *coid*) used by the QNX Neutrino-specific functions.

Errors:**EACCES**

Search permission is denied on a component of the *path* prefix, or the file exists and the permissions specified by *oflag* are denied, or the file doesn't exist and write permission is denied for the parent directory of the file to be created.

EBADFSYS

While attempting to open the named file, either the file itself or a component of the path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You'll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY

File access was denied due to a conflicting open (see [sopen\(\)](#) (p. 3083)).

EEXIST

The `O_CREAT` and `O_EXCL` flags are set, and the named file exists.

EINTR

The *open()* operation was interrupted by a signal.

EINVAL

The requested synchronized modes (`O_SYNC`, `O_DSYNC`, `O_RSYNC`) aren't supported.

EISDIR

The named file is a directory, and the *oflag* argument specifies write-only or read/write access.

ELOOP

Too many levels of symbolic links or prefixes, including these cases:

- The flags include `O_NOFOLLOW`, and the last component of the path is a symbolic link.
- The flags include `O_NOSYMLINK`, and any component of the path is a symbolic link.

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the *path* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENFILE

Too many files are currently open in the system.

ENOENT

The `O_CREAT` flag isn't set, and the named file doesn't exist; or `O_CREAT` is set and either the path prefix doesn't exist, or the *path* argument points to an empty string.

ENOSPC

The directory or filesystem that would contain the new file can't be extended.

ENOSYS

The *open()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix isn't a directory.

ENXIO

The `O_NONBLOCK` flag is set, the named file is a FIFO, `O_WRONLY` is set, no process has the file open for reading, or the media associated with the file has been removed (e.g. CD, floppy).

EOVERFLOW

The named file is a regular file and the size of the file can't be represented correctly in an object of type `off_t`.

EROFS

The named file resides on a read-only filesystem and either `O_WRONLY`, `O_RDWR`, `O_CREAT` (if the file doesn't exist), or `O_TRUNC` is set in the *oflag* argument.

Examples:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    /* open a file for output          */
    /* replace existing file if it exists */
    /* with read/write perms for owner  */
    fd = open( "myfile.dat",
               O_WRONLY | O_CREAT | O_TRUNC,
               S_IRUSR | S_IWUSR );

    /* read a file that is assumed to exist */

    fd = open( "myfile.dat", O_RDONLY );

    /* append to the end of an existing file */
    /* write a new file if file doesn't exist */
    /* with full read/write permissions      */
    fd = open( "myfile.dat",
               O_WRONLY | O_CREAT | O_APPEND,
               S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP
               | S_IROTH | S_IWOTH );
    return EXIT_SUCCESS;
}
```

Classification:

`open()` is [POSIX 1003.1](#) (p. 46); `open64()` is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The *open()* function includes POSIX 1003.1-1996 and QNX Neutrino extensions.

opendir()

Open a directory

Synopsis:

```
#include <dirent.h>

DIR * opendir( const char * dirname );
```

Arguments:

dirname

The path of the directory to be opened. It can be relative to the current working directory, or an absolute path.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `opendir()` function is used with [readdir\(\)](#) (p. 2514) and [closedir\(\)](#) (p. 338) to get the list of file names contained in the directory specified by *dirname*.

You can read more than one directory at the same time using the `opendir()`, [readdir\(\)](#) (p. 2514), [rewinddir\(\)](#) (p. 2648) and [closedir\(\)](#) (p. 338) functions.



The result of using a directory stream after one of the `exec*()` or `spawn*()` functions is undefined. After a call to the [fork\(\)](#) (p. 685) function, either the parent *or* the child (but not both) can continue processing the directory stream using [readdir\(\)](#) and [rewinddir\(\)](#). If both the parent and child processes use these functions, the result is undefined. Either process can use [closedir\(\)](#).

Returns:

A pointer to a `DIR` structure required for subsequent calls to [readdir\(\)](#) to retrieve the file names in *dirname*, or `NULL` if *dirname* isn't a valid path ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission is denied for a component of *dirname*, or read permission is denied for *dirname*.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of *dirname* exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.

ENOENT

The named directory doesn't exist.

ENOSYS

The *opendir()* function isn't implemented for the filesystem specified in *dirname*.

ENOTDIR

A component of *dirname* isn't a directory.

Examples:

Get a list of files contained in the directory `/home/fred`:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main( void )
{
    DIR* dirp;
    struct dirent* direntp;

    dirp = opendir( "/home/fred" );
    if( dirp == NULL ) {
        perror( "can't open /home/fred" );
    } else {
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;

            printf( "%s\n", direntp->d_name );
        }

        closedir( dirp );
    }

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

openfd()

Open for private access a file associated with a given descriptor

Synopsis:

```
#include <unistd.h>

int openfd( int fd,
            int oflag );
```

Arguments:

fd

A file descriptor associated with the file that you want to open.

oflag

How you want to open the file; a combination of the following bits:

- **O_RDONLY** — permit the file to be only read.
- **O_WRONLY** — permit the file to be only written.
- **O_RDWR** — permit the file to be both read and written.
- **O_APPEND** — cause each record that's written to be written at the end of the file.
- **O_TRUNC** — truncate the file to contain no data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *openfd()* function opens the file associated with the file descriptor, *fd*. This is similar to [dup\(\)](#) (p. 483), except the new *fd* has private access modes and offset. The access mode, *oflag*, must be equal to or more restrictive than the access mode of the source *fd*.

Returns:

A file descriptor, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid file descriptor *fd*.

EACCES

The access mode specified by *oflag* isn't equal to or more restrictive than the access mode of the source *fd*.

EBUSY

Sharing mode (*sflag*) was denied due to a conflicting open (see [sopenfd\(\)](#) (p. 3087)).

Examples:

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main ( void )
{
    int fd, fd2, oflag;

    fd = open ("/etc/passwd", O_RDONLY);
    fd2 = openfd ( fd, O_RDONLY );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

openlog()

Open the system log

Synopsis:

```
#include <syslog.h>

void openlog( const char * ident,
              int logopt,
              int facility );
```

Arguments:

ident

A string that you want to prepend to every message.

logopt

A bit field specifying logging options; a combination of one or more of the following values with an OR operation:

LOG_CONS

If *syslog()* can't pass the message to *syslogd*, it attempts to write the message to the */dev/console* device. The */dev/console* device is usually a symlink (see the *ln* command) to a real device (e.g. */dev/text*, */dev/con1* or */dev/ser1*).

LOG_NDELAY

Open the connection to *syslogd* immediately. Normally the opening is delayed until the first message is logged.

LOG_PERROR

Write the message to standard error output as well to the system log.

LOG_PID

Log the process ID with each message. This is useful for identifying instantiations of daemons.

facility

Encode a default facility to be assigned to all messages that don't have an explicit facility encoded. In the following list, parameter values marked with an asterisk (*) aren't used by any of the QNX Neutrino standard utilities.

LOG_AUTH *

Authorization system.

LOG_AUTHPRIV *

Same as LOG_AUTH, but logged to a file readable only by selected individuals.

LOG_CRON *

Clock daemon.

LOG_DAEMON

System daemons (such as `routed`) that aren't explicitly provided for by other facilities.

LOG_FTP

File transfer protocol daemon.

LOG_KERN *

Messages generated by the kernel. These can't be generated by any user processes.

LOG_LPR

Line printer spooling system.

LOG_MAIL

Mail system.

LOG_NEWS *

Network news system.

LOG_SYSLOG

Messages generated internally by `syslogd`.

LOG_USER*

Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_UUCP *

The uucp system.

LOG_LOCAL0 through LOG_LOCAL7 *

Reserved for local use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `openlog()` function opens the system log and provides for more specialized processing of the messages sent by [syslog\(\)](#) (p. 3316) and [vsyslog\(\)](#) (p. 3643).

Examples:

See [syslog\(\)](#) (p. 3316).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

openpty()

Find an available pseudo-tty

Synopsis:

```
#include <unix.h>

int openpty( int* amaster,
             int* aslave,
             char* name,
             struct termios* termp,
             struct winsize* winp );
```

Arguments:

amaster

A pointer to a location where *openpty()* can store the file descriptor of the master side of the pseudo-tty.

aslave

A pointer to a location where *openpty()* can store the file descriptor of the slave side of the pseudo-tty.

name

NULL, or a pointer to a buffer where *openpty()* can store the filename of the slave side of the pseudo-tty.

termp

NULL, or a pointer to a [termios](#) (p. 3379) structure that describes the terminal's control attributes to apply to the slave side of the pseudo-tty.

winp

A pointer to a `winsize` structure that defines the window size to use for the slave side of the pseudo-tty.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *openpty()* function finds and opens an available pseudo-tty.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**ENOENT**

There are no ttys available.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

out8()

Write an 8-bit value to a port

Synopsis:

```
#include <hw/inout.h>

void out8( uintptr_t port,
           uint8_t val );
```

Arguments:

port

The port you want to write the value to.

val

The value that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out8()` function writes an 8-bit value, specified by *val*, to the specified *port*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [proc_mgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

out8s()

Write 8-bit values to a port

Synopsis:

```
#include <hw/inout.h>

void * out8s( const void * buff,
              unsigned len,
              uintptr_t port );
```

Arguments:

val

A pointer to a buffer that holds the values that you want to write.

len

The number of values that you want to write.

port

The port you want to write the values to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *out8s()* function writes *len* 8-bit values from the buffer pointed to by *buff* to the specified *port*.

Returns:

A pointer to the end of the written data.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

out16(), outbe16(), outle16()

Write a 16-bit value to a port

Synopsis:

```
#include <hw/inout.h>

void out16( uintptr_t port,
            uint16_t val );

#define outbe16( port,
                val ) ...

#define outle16( port,
                val ) ...
```

Arguments:

port

The port you want to write the value to.

val

The value that you want to write.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *out16()* function writes the native-endian 16-bit value, specified by *val*, to the specified *port*.

The *outbe16()* and *outle16()* macros write the native-endian 16-bit value, specified by *val*, to the specified *port* in big-endian or little-endian format, respectively.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

`outbe16()` and `outle16()` are implemented as macros.

out16s()

Write words to a port

Synopsis:

```
#include <hw/inout.h>

void * out16s( const void * buff,
               unsigned len,
               uintptr_t port );
```

Arguments:

val

A pointer to a buffer that holds the values that you want to write.

len

The number of values that you want to write.

port

The port you want to write the values to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out16s()` function writes *len* words from the buffer pointed to by *buff* to the specified *port*.

Returns:

A pointer to the end of the written data.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROCmgr_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

out32(), outbe32(), outle32()

Write a 32-bit value to a port

Synopsis:

```
#include <hw/inout.h>

void out32( uintptr_t port,
           uint32_t val );

#define outbe16( port,
                val ) ...

#define outle32( port,
                val ) ...
```

Arguments:

port

The port you want to write the value to.

val

The value that you want to write.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *out32()* function writes the 32-bit value, specified by *val*, to the specified *port*.

The *outbe32()* and *outle32()* functions macros the native-endian 32-bit value, specified by *val*, to the specified *port* in big-endian or little-endian format, respectively.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROCmgr_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

`outbe16()` and `outle16()` are implemented as macros.

out32s()

Write longs to a port

Synopsis:

```
#include <hw/inout.h>

void * out32s( const void * buff,
               unsigned len,
               uintptr_t port );
```

Arguments:

- val**
A pointer to a buffer that holds the values that you want to write.
- len**
The number of values that you want to write.
- port**
The port you want to write the values to.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The out32s() function writes len longs from the buffer pointed to by buff to the specified port.

Returns:

A pointer to the end of the written data.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The calling thread must have the `PROC_MGR_AID_IO` ability enabled and must have I/O privileges; see [procmgr_ability\(\)](#) (p. 2141), and [ThreadCtl\(\)](#) (p. 3408)'s `_NTO_TCTL_IO` command for details.

The calling process must also use [mmap_device_io\(\)](#) (p. 1666) to access the device's I/O registers.

Chapter 18

P

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>PCI_IO_ADDR()</i>	pci_attach_device() (p. 1966)
<i>PCI_IS_IO()</i>	pci_attach_device() (p. 1966)
<i>PCI_IS_MEM()</i>	pci_attach_device() (p. 1966)
<i>PCI_MEM_ADDR()</i>	pci_attach_device() (p. 1966)
<i>PCI_ROM_ADDR()</i>	pci_attach_device() (p. 1966)
<i>ppsparse()</i>	Persistent Publish/Subscribe <i>Developer's Guide</i>
<i>PTHREAD_BARRIER_INITIALIZER()</i>	pthread_barrier_init() (p. 2239)

This chapter describes the API components whose names begin with “P”.

pathconf()

Return the value of a configurable limit

Synopsis:

```
#include <unistd.h>

long pathconf( const char* path,
               int name );
```

Arguments:

path

The name of the file whose limit you want to get.

name

The name of the configurable limit; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pathconf()* function returns a value of a configurable limit indicated by *name*, which is associated with the filename given in *path*.

Configurable limits are defined in `<confname.h>`, and include at least the following values:

`_PC_ASYNC_IO`

Defined if asynchronous I/O is supported for the file.

`_PC_CHOWN_RESTRICTED`

If defined (not -1), indicates that the use of the [chown\(\)](#) (p. 292) function is restricted to a process with root privileges, and to changing the group ID of a file to the effective group ID of the process or to one of its supplementary group IDs.

`_PC_LINK_DIR`

Defined (not -1) if the filesystem permits the unlinking of a directory.

`_PC_LINK_MAX`

Maximum value of a file's link count.

`_PC_MAX_CANON`

Maximum number of bytes in a terminal's canonical input buffer (edit buffer).

`_PC_MAX_INPUT`

Maximum number of bytes in a terminal's raw input buffer.

`_PC_NAME_MAX`

Maximum number of bytes in a file name (not including the terminating null).

`_PC_NO_TRUNC`

If defined (not -1), indicates that the use of pathname components longer than the value given by `_PC_NAME_MAX` will generate an error.

`_PC_PATH_MAX`

Maximum number of bytes in a pathname (not including the terminating null).

`_PC_PIPE_BUF`

Maximum number of bytes that can be written atomically when writing to a pipe.

`_PC_PRIO_IO`

Defined (not -1) if prioritized I/O is supported for the file.

`_PC_SYNC_IO`

Defined (not -1) if synchronous I/O is supported for the file.

`_PC_VDISABLE`

If defined (not -1), this is the character value which can be used to individually disable special control characters in the [termios](#) (p. 3379) control structure.

Returns:

The requested configurable limit, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission is denied for a component of *path*.

EINVAL

The *name* argument is invalid, or the indicated limit isn't supported.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The *path* argument, or a component of *path*, is too long.

ENOENT

The file doesn't exist.

ENOSYS

The *pathconf()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix isn't a directory.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    long value;

    value = pathconf( "/dev/con1", _PC_MAX_INPUT );
    printf( "Input buffer size is %ld bytes\n",
           value );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pathfind(), pathfind_r()

Search for a file in a list of directories

Synopsis:

```
#include <libgen.h>

char *pathfind( const char *path,
                const char *name,
                const char *mode );

char *pathfind_r( const char *path,
                  const char *name,
                  const char *mode,
                  char *buff,
                  size_t buff_size );
```

Arguments:***path***

A string that specifies the list of the directories that you want to search. The directories named in *path* are separated by colons.

name

The name of the file you're looking for. If *name* begins with a slash, the name is treated as an absolute pathname, and *path* is ignored.

mode

A string of option letters chosen from:

r

Readable.

w

Writable.

x

Executable.

f

Normal file.

b

Block special.

c

Character special.

d

Directory.

p

FIFO (pipe).

u

Set user ID bit.

g

Set group ID bit.

k

Sticky bit.

s

Size nonzero.

buff

(*pathfind_r()* only) A pointer to a buffer where *pathfind_r()* can store the path of the file found.

buff_size

(*pathfind_r()* only) The size of the buffer that *buff* points to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pathfind()* function searches the directories named in *path* for the file *name*. The *pathfind_r()* function is a thread-safe version of *pathfind()*.

Options read, write, and execute are checked relative to the real (not the effective) user ID and group ID of the current process.

If the file *name*, with all the characteristics specified by *mode*, is found in any of the directories specified by *path*, then these functions return a pointer to a string containing the member of *path*, followed by a slash character (/), followed by *name*.

An empty path member is treated as the current directory. If *name* is found in the current directory, a slash isn't prepended to it; the unadorned name is returned.

The *pathfind_r()* also includes a buffer, *buff*, and its size, *buff_size*. This buffer is used to hold the path of the file found.

Returns:

The path found, or NULL if the file couldn't be found.

Examples:

Find the `ls` command using the ***PATH*** environment variable:

```
pathfind (getenv ("PATH"), "ls", "rx");
```

Classification:

[Unix](#) (p. 46)

Table 9: *pathfind()*

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Table 10: *pathfind_r()*

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The string pointed to by the returned pointer is stored in an area that's reused on subsequent calls to *pathfind()*. Don't free this string.

Use *pathfind_r()* in multithreaded applications.

pathmgr_symlink()

Create a symlink

Synopsis:

```
#include <sys/pathmgr.h>

int pathmgr_symlink( const char * path,
                    const char * symlink );
```

Arguments:

path

The path that you want to link to.

symlink

The name of the link that you want to create.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pathmgr_symlink()` function creates a symbolic link, `path`, in the process manager that redirects to the path specified by `symlink`.

In order to create the link, your process must have the `PROC_MGR_AID_PATHSPACE` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The [pathmgr_unlink\(\)](#) (p. 1945) function removes the link.



- The symbolic link isn't permanent and is lost when the system reboots.
- You can get some surprising results if you create a symbolic link to a directory that's a symbolic link to somewhere else. For more information, see “Symbolic links” in the Working with Filesystems chapter of the QNX Neutrino *User's Guide*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <sys/pathmgr.h>

int main(int argc, char **argv) {
    /* Create a link from /mytmp to /dev/shmem */
    if(pathmgr_symlink("/dev/shmem", "/mytmp") == -1) {
        perror("Can't make link");
    }

    getchar();
    if(pathmgr_unlink("/mytmp") == -1) {
        perror("Can't unlink ");
    }

    return 0;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pathmgr_unlink()

Remove a link

Synopsis:

```
#include <sys/pathmgr.h>

int pathmgr_unlink( const char * path );
```

Arguments:

path

The link that you want to remove.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *pathmgr_unlink()* function removes the link created by [pathmgr_symlink\(\)](#) (p. 1943).

Returns:

0

Success.

-1

An error occurred.

Examples:

See [pathmgr_symlink\(\)](#) (p. 1943).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pause()

Suspend the calling thread until delivery of a signal

Synopsis:

```
#include <unistd.h>

int pause( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pause()* function suspends the calling thread until delivery of a signal whose action is either to execute a signal handler or to terminate the process.

If the action is to terminate the process, *pause()* doesn't return. If the action is to execute a signal handler, *pause()* returns after the signal handler returns.

Returns:

On error, *pause()* returns -1 and sets [errno](#) (p. 539) to `EINTR`; otherwise, it never returns.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    /* set an alarm to go off in 5 seconds */
    alarm( 5 );

    /*
     * Wait until we receive a SIGALRM signal. However,
     * since we don't have a signal handler, any signal
     * will kill us.
     */
    printf( "Hang around, "
           "waiting to die in 5 seconds\n" );
    pause();
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_arm()

Arm the devp-pccard server

Synopsis:

```
#include <sys/pccard.h>

int pccard_arm( pccard_t handle,
               int devtype,
               unsigned event,
               int coid );
```

Arguments:

handle

The handle returned by [pccard_attach\(\)](#) (p. 1952).

devtype

The type of device that your application wants to be informed about. Valid devices are:

- `_PCCARD_DEV_AIMS` — Auto Incrementing Mass Storage.
- `_PCCARD_DEV_ALL` — all devices.
- `_PCCARD_DEV_FIXED_DISK` — any hard drive.
- `_PCCARD_DEV_GPIB` — General Purpose Interface Bus card.
- `_PCCARD_DEV_MEMORY` — memory type device.
- `_PCCARD_DEV_NETWORK` — any network adapter.
- `_PCCARD_DEV_PARALLEL` — PC parallel device.
- `_PCCARD_DEV_SCSI` — any SCSI interface.
- `_PCCARD_DEV_SERIAL` — 16450 serial device.
- `_PCCARD_DEV_SOUND` — any sound adapter.
- `_PCCARD_DEV_VIDEO` — any video adapter.

event

The type of event that you want to be notified of:

- `_PCCARD_ARM_INSERT_REMOVE` — card insertion/removal.

coid

A connection ID, obtained from [ConnectAttach\(\)](#) (p. 351), that's used to send the pulse.

Library:

libpccard

Use the `-l pccard` option to `qcc` to link against this library.

Description:

The `pccard_arm()` function call is used to request that the `devp-pccard` server notify the user application, via a pulse, when the specified event occurs.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid *handle* parameter.

Examples:

```
/*
 * Ask to be informed when a Network card is inserted
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/neutrino.h>
#include <sys/pccard.h>

int main (void)
{
    pccard_t    handle;
    int         chid, coid;
    char        buf [10];
    struct _pccard_info io;

    if ((handle = pccard_attach (0)) == -1) {
        printf ("Unable to attach to PCCARD\n");
        exit (EXIT_FAILURE);
    }

    if ((chid = ChannelCreate (_NTO_CHF_FIXED_PRIORITY)) == -1) {
        printf ("Unable to create channel\n");
        exit (EXIT_FAILURE);
    }
}
```

```

if ((coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL,
                          0)) == -1) {
    printf ("Unable to ConnectAttach\n");
    exit (EXIT_FAILURE);
}

if (pccard_arm (handle, _PCCARD_DEV_NETWORK,
               _PCCARD_ARM_INSERT_REMOVE, coid) == -1) {
    perror ("Arm failed");
    exit (EXIT_FAILURE);
}

/* To be informed about any card insertion/removal event,
 * change _PCCARD_DEV_NETWORK to _PCCARD_DEV_ALL.
 */

/*
 * MsgReceive (chid, ....);
 * Other user logic...
 */

/* Get information from socket 0 - function 0 */
if (pccard_info (handle, 0, &io, sizeof (io)) == -1) {
    perror ("Info failed");
    exit (EXIT_FAILURE);
}
if (io.flags & _PCCARD_FLAG_CARD) {
    printf ("Card inserted in socket 1 - Type %x\n",
           io.window [0].device & 0xff00);
}
/* Now lock the card in socket 1 with exclusive access */
if (pccard_lock (handle, 0, 0, 0_RDWR | 0_EXCL) == -1) {
    perror ("Lock failed");
    exit (EXIT_FAILURE);
}
/* Read 2 bytes of the CIS from offset 0 in attribute memory */
if (pccard_raw_read (handle, 0, _PCCARD_MEMTYPE_ATTRIBUTE,
                    0, 2, buf) == -1) {
    perror ("Raw read");
    exit (EXIT_FAILURE);
}
/* More user logic... */
}
pccard_unlock (handle, 0, 0);
pccard_detach (handle);

return (EXIT_SUCCESS);
}

```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_attach()

Attach to the devp-pccard server

Synopsis:

```
#include <sys/pccard.h>
pccard_t pccard_attach( int reserved );
```

Arguments:

reserved

Pass 0 for this argument.

Library:

libpccard

Use the -l pccard option to gcc to link against this library.

Description:

The *pccard_attach()* function attaches a user application to the devp-pccard server. You must call this function before using any of the other PC card functions, because it returns a handle that all the other PC Card functions use.

Returns:

>0

A value to be used as *handle* in all other PC Card function calls.

-1

Can't locate the devp-pccard server.

-2

Send to devp-pccard server failed.

-3

The devp-pccard server returned an error ([errno](#) (p. 539) is set).

Errors:

EBUSY

The devp-pccard server is unable to service this request.

Examples:

See [pccard_arm\(\)](#) (p. 1950).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_detach()

Detach from the devp-pccard server

Synopsis:

```
#include <sys/pccard.h>

int pccard_detach( pccard_t handle );
```

Arguments:

handle

The handle returned by [pccard_attach\(\)](#) (p. 1952).

Library:

libpccard

Use the -l pccard option to qcc to link against this library.

Description:

The *pccard_detach()* function detaches the user application from the devp-pccard server. Any locks that you previously applied with [pccard_lock\(\)](#) (p. 1958) are freed.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid *handle* parameter.

Examples:

See [pccard_arm\(\)](#) (p. 1950).

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_info()

Obtain socket information from the devp-pccard server

Synopsis:

```
#include <sys/pccard.h>

int pccard_info( pccard_t handle,
                 int socket,
                 struct _pccard_info* info,
                 unsigned size );
```

Arguments:

handle

The handle returned by [pccard_attach\(\)](#) (p. 1952).

socket

Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.

info

A pointer to a `_pccard_info` structure that the function fills with the socket information. For more information, see [below](#) (p. 1956).

size

Size of the `_pccard_info` structure.

Library:

`libpccard`

Use the `-l pccard` option to `qcc` to link against this library.

Description:

The `pccard_info()` function call retrieves socket setup information from the devp-pccard server. The information is returned in the `_pccard_info` structure.

`_pccard_info` structure

The `_pccard_info` structure is defined in `<pccard.h>` as:

```
struct _pccard_info {
    int16_t      socket;           // Socket number (0 based)
```



```

uint16_t      status;           // Card status (from socket services spec)
uint32_t      flags;           // Flags (_PCCARD_FLAG_*)
uint8_t       vcc;             // Current Vcc (in tenths of volts)
uint8_t       vpp;            // Current Vpp (in tenths of volts)
uint8_t       num_windows;     // Number of windows described below
uint8_t       index;          // Index for CardBus devices
uint16_t      manufacturer;    // Manufacturer ID from PCCARD
uint16_t      card_type;       // Card Type from PCCARD
uint16_t      device_id;       // CardBus device id
uint16_t      vendor_id;       // CardBus vendor id
uint16_t      busnum;          // PCI bus number
uint16_t      devfuncnum;      // PCI device and function number
struct _pccard_window {
    uint16_t      window;       // Window type (_PCCARD_WINDOW_*)
    uint16_t      flags;        // Window flags (_PCCARD_WINFLAG_*)
    mpid_t        pid;          // Locking pid
    uint16_t      device;       // Device type (_PCCARD_DEV_*)
    uint16_t      dummy;
    uint32_t      dev_size;     // Size of memory device
    uint32_t      reserved3;
    union {
        struct _pccard_irq {
            uint16_t      flags; // (_PCCARD_IRQFLAG_*)
            uint16_t      irq;
        };
        struct _pccard_memio {
            uint32_t      base;   // Base address (in host address space)
            uint32_t      size;   // Size of window
            uint32_t      offset; // offset of region from base of card
            uint16_t      flags;  // (_PCCARD_MEMIOFLAG_*)
            uint16_t      dummy2;
        };
    };
} un;
} window[_PCCARD_MAX_WINDOWS];
};

```

Returns:**A positive integer**

Success. The *socket* parameter is returned.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**ENODEV**

Invalid *socket* parameter.

Examples:

See [pccard_arm\(\)](#) (p. 1950).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_lock()

Lock the window of the card in the selected socket

Synopsis:

```
#include <sys/pccard.h>

int pccard_lock( pccard_t handle,
                 int socket,
                 int index,
                 int oflag );
```

Arguments:

handle

The handle returned by [pccard_attach\(\)](#) (p. 1952).

socket

Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.

index

The window/function number that you want to lock. You can get the window number from the `_pccard_info` structure (see [pccard_info\(\)](#) (p. 1956)).

oflag

Created by ORing the values required (e.g. `O_RDWR | O_EXCL`) for read/write and exclusive access.

Library:

`libpccard`

Use the `-l pccard` option to `qcc` to link against this library.

Description:

The `pccard_lock()` function call provides exclusive or shared access to the PC Card in *socket* and also sets access permissions.

Returns:

A positive integer

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid *handle* parameter.

EBUSY

The window is already locked by another process.

ENODEV

Invalid *socket* parameter, no PC Card is present in the socket, or invalid *index* parameter.

Examples:

See [pccard_arm\(\)](#) (p. 1950).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_raw_read()

Read the raw CIS data from the PC Card

Synopsis:

```
#include <sys/pccard.h>

ssize_t pccard_raw_read( pccard_t handle,
                        int socket,
                        int type,
                        unsigned addr,
                        ssize_t len,
                        void* buf );
```

Arguments:

handle

The handle returned by [*pccard_attach\(\)*](#) (p. 1952).

socket

Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.

type

The type of memory that you want to read. Valid values are:

- `_PCCARD_MEMTYPE_COMMON`
- `_PCCARD_MEMTYPE_ATTRIBUTE`

addr

The memory address that you want to read from the CIS.

len

The size of the memory that you want to read.

buf

A pointer to a buffer where the function can store the information that it reads from the PC Card.

Library:

libpccard

Use the -l pccard option to gcc to link against this library.

Description:

The *pccard_raw_read()* function returns the raw CIS (Card Information Structure) data from the PC Card.

Returns:**A positive integer**

Success. The length read is returned.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid *handle* parameter.

ENODEV

Invalid *socket* parameter.

Examples:

See [pccard_arm\(\)](#) (p. 1950).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pccard_unlock()

Unlock the window of the card in the selected socket

Synopsis:

```
#include <sys/pccard.h>

int pccard_unlock( pccard_t handle,
                  int socket,
                  int index );
```

Arguments:

handle

The handle returned by [pccard_attach\(\)](#) (p. 1952).

socket

Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.

index

The window/function number that you want to unlock. You can get the window number from the `_pccard_info` structure (see [pccard_info\(\)](#) (p. 1956)).

Library:

`libpccard`

Use the `-l pccard` option to `gcc` to link against this library.

Description:

The `pccard_unlock()` function unlocks a window previously locked by a call to [pccard_lock\(\)](#) (p. 1958). It can only unlock a window locked by the same process ID — you can't unlock a window locked by another process.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid *handle* parameter.

ENODEV

Invalid *socket* parameter, no PC Card is present in the socket, or invalid *index* parameter.

Examples:

See [pccard_arm\(\)](#) (p. 1950).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_attach()

Connect to the PCI server

Synopsis:

```
#include <hw/pci.h>

int pci_attach( unsigned flags );
```

Arguments:

flags

There are currently no flags defined for this function.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *pci_attach()* function connects to the Peripheral Component Interconnect (PCI) server.



You must call *pci_attach()* before calling any of the other PCI functions.

Returns:

A handle used for calling *pci_detach()* (p. 1975), or -1 if an error occurs.

Errors:

See *open()* (p. 1905).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_attach_device()

Attach a driver to a PCI device

Synopsis:

```
#include <hw/pci.h>

void* pci_attach_device(
    void* handle,
    uint32_t flags,
    uint16_t idx,
    struct pci_dev_info* info );
```

Arguments:

handle

A handle that identifies the device. The first time you call this function, set *handle* to NULL. This function returns a handle that you can use in a subsequent call to allocate resources for the device.

flags

Flags that tell the PCI server how you want it to handle resources, which resources to scan for, and which resources to allocate; see “[Flags](#) (p. 1969),” below.

idx

The index of the device: 0 for the first device, 1 for the second, and so on.

info

A pointer to a [pci_dev_info](#) (p. 1967) structure (see below) that specifies the class code, vendor/device ID, or bus number and device/function number that you want to scan for. The function fills in this structure with information about the device.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_attach_device()* function attaches a driver to a PCI device.



You must successfully call *pci_attach()* (p. 1964) before calling any of the other PCI functions.

Typically drivers use this function to attach themselves to a PCI device, so that other drivers can't attach to the same device. If you specify the `PCI_SHARE` flag (see “*Flags*” (p. 1969),” below), then multiple drivers can attach to the same device.

The server can scan based on a class code, vendor/device ID, or bus number and device/function number. To control the server scanning, initialize the appropriate fields of the *info* structure and set the appropriate flags.

When you first attach to an uninitialized device, the PCI server assigns all the I/O ports, memory and IRQs required for the device. It also does the IRQ routing for you. Once this has completed successfully, it fills in all these values into your *pci_dev_info* structure to return these values to your application.

When a driver attaches to a device, the PCI server allocates the necessary resources for the device from *procnto* using the *rsrddbmgr** calls. On X86 BIOS systems, these resources are normally allocated by the BIOS, but on non-x86 systems, these resources have to be allocated from *procnto*.

You can detach the device by passing its handle to *pci_detach_device()* (p. 1977). If you call *pci_detach()* (p. 1975), any resources that *pci_attach_device()* allocates are freed.

pci_dev_info structure

This function fills in a *pci_dev_info* structure that describes an occurrence of a device.



The *pci_attach_device()* function doesn't map any of the I/O or memory regions into the process's address space. The addresses returned in the *pci_dev_info* structure are all physical addresses.

This structure has the following members:

uint16_t DeviceId

The device ID (input/output). For a list of supported device IDs, see `<hw/pci_devices.h>`.

uint16_t VendorId

The vendor ID (input/output). For a list of supported vendor IDs, see `<hw/pci_devices.h>`.

uint16_t SubsystemId

The subsystem ID (output).

uint16_t *SubsystemVendorId*

The subsystem vendor ID (output).

uint8_t *BusNumber*

The bus number (input/output).

uint8_t *DevFunc*

The device/function number (input/output).

uint8_t *Revision*

The device revision (output).

uint32_t *Class*

The class code (input/output). For a list of class codes, see `<hw/pci.h>`. This field is an ORed combination of a class code and a subclass code (e.g. `PCI_CLASS_DISPLAY | PCI_SUBCLASS_DISPLAY_XGA`).

uint32_t *Irq*

The interrupt number (output).

uint64_t *CpuIoTranslation*

The CPU-to-PCI translation value ($pci\_addr = cpu\_addr - translation$).

uint64_t *CpuMemTranslation*

The CPU-to-PCI memory translation ($pci\_addr = cpu\_addr - translation$).

uint64_t *CpuIsaTranslation*

The CPU-to-ISA memory translation ($pci\_addr = cpu\_addr - translation$).

uint64_t *CpuBmstrTranslation*

The translation from the CPU busmaster address to the PCI busmaster address ($pci\_addr = cpu\_addr + translation$).

uint64_t *PciBaseAddress* [6]

The PCI base address (array of six `uint64_t` items).



This function decodes bits 1 and 2 to see whether the register is 32 or 64 bits wide, hence the 64-bit values for the base registers.

uint64_t CpuBaseAddress [6]

The CPU base address (an array of six uint64_t items).

Some platforms translate addresses across PCI bridges, so that there's one address on the PCI side of the bridge and another on the CPU side. Under x86, the *PciBaseAddress* and *CpuBaseAddress* are the same, but under other platforms, these will be different. In your user application you should always use the *CpuBaseAddress*.

uint32_t BaseAddressSize [6]

The size of the base address aperture into the board (an array of six uint32_t items).

uint64_t PciRom

The PCI ROM address.

uint64_t CpuRom

The CPU ROM address.

uint32_t RomSize

The size of the aperture into the board.

Flags

The *flags* parameter tells the PCI server how resources are to be handled, which resources to scan for, and which resources to allocate. If you pass 0 for the flags, the default is `PCI_SEARCH_VENDEV`.

These bits control how resources are handled:

PCI_SHARE

Allow resources to be shared with other drivers. If this isn't set, no other driver can attach to the device.

PCI_PERSIST

Resources persist after the device is detached.

The following bits ask the PCI server to scan for a device based on the fields that you specified in the structure pointed to by *info*:

PCI_SEARCH_VEND

VendorID

PCI_SEARCH_VENDEV

DeviceId and VendorId

PCI_SEARCH_CLASS

Class

PCI_SEARCH_BUSDEV

BusNumber and DevFunc

These bits specify which members of the structure the server should initialize:

PCI_INIT_IRQ

Irq

PCI_INIT_ROM

PciRom and CpuRom

PCI_INIT_BASE0 ... PCI_INIT_BASE5

The specified entries of the *PciBaseAddress* and *CpuBaseAddress* arrays

PCI_INIT_ALL

All members except *PciRom* and *CpuRom*

The bits also include:

PCI_MASTER_ENABLE

Enable bus mastering on the device.

PCI_USE_MSI

Use Message Signaled Interrupts (MSIs) instead of the default IRQs.

PCI_USE_MSIX

Use Extended Message Signaled Interrupts (MSI-Xs) instead of the default IRQs.

To determine if a device supports MSI or MSI-X, run the `pci -v` command and look for MSI and MSI-X in the device's capabilities. To do this programatically, you have to traverse the device's capabilities structure (see the “Examples” section below).



- In order to use MSI or MSI-X, you need to run `startup-apic` and `pci-bios-v2`.
 - If a device can use both MSI and MSI-X interrupts, and you specify the `PCI_USE_MSI` flag, then only MSI interrupts are allocated.
-

Testing and converting addresses

To facilitate the testing of addresses returned by the PCI server, at least the following macros are defined in the `<hw/pci.h>` header file:

PCI_IS_IO(address)

Test whether the address is an I/O address.

PCI_IS_MEM(address)

Test whether the address is a memory address.

PCI_IO_ADDR(address)

Convert the address returned by the PCI server to an I/O address.

PCI_MEM_ADDR(address)

Convert the address returned by the PCI server to a memory address.

PCI_ROM_ADDR(address)

Convert the address returned by the PCI server to a ROM address.

For example:

```
{
    uint64_t    port;

    /* Test the address returned by the pci server */
    if (PCI_IS_IO(addr))
        port = (PCI_IO_ADDR(addr));
}
```

Returns:

A handle to be used for other *pci_** calls associated with a handle, or NULL if an error occurs (*errno* (p. 539) is set).

Errors:

EBUSY

An application has already attached to the device. If it's safe to share the device, specify `PCI_SHARE` in the *flags* field.

EINVAL

The function couldn't attach a resource to the device.

ENODEV

This device wasn't found.

Examples:

Attach to and allocate all resources for the first occurrence of an Adaptec 2940 adapter:

```
#include <hw/pci.h>
#include <hw/pci_devices.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int pidx;
    void* hdl;
    int phdl;
    struct pci_dev_info inf;

    /* Connect to the PCI server */
    phdl = pci_attach( 0 );
    if( phdl == -1 ) {
        fprintf( stderr, "Unable to initialize PCI\n" );

        return EXIT_FAILURE;
    }

    /* Initialize the pci_dev_info structure */
    memset( &inf, 0, sizeof( inf ) );
    pidx = 0;
    inf.VendorId = PCI_VENDOR_ID_ADAPTEC;
    inf.DeviceId = PCI_DEVICE_ID_ADAPTEC_2940F;

    hdl = pci_attach_device( NULL, PCI_INIT_ALL, pidx, &inf );
    if( hdl == NULL ) {
        fprintf( stderr, "Unable to locate adapter\n" );
    } else {
        /* Do something to the adapter */
        pci_detach_device( hdl );
    }

    /* Disconnect from the PCI server */
    pci_detach( phdl );

    return EXIT_SUCCESS;
}
```

Attach to the first occurrence of an Adapter 2940 adapter and allocate resources in a second call:

```
#include <hw/pci.h>
#include <hw/pci_devices.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int pidx;
    void* hdl;
    void* retval;
    int phdl;
    struct pci_dev_info inf;

    phdl = pci_attach( 0 );
    if( phdl == -1 ) {
        fprintf( stderr, "Unable to initialize PCI\n" );

        return EXIT_FAILURE;
    }

    memset( &inf, 0, sizeof( inf ) );
    pidx = 0;
    inf.VendorId = PCI_VENDOR_ID_ADAPTEC;
    inf.DeviceId = PCI_DEVICE_ID_ADAPTEC_2940F;

    hdl = pci_attach_device( NULL, 0, pidx, &inf );
    if( hdl == NULL ) {
        fprintf( stderr, "Unable to locate adapter\n" );
    }
}
```



```

    retval = pci_attach_device( hdl, PCI_INIT_ALL, pidx, &inf );
    if( retval == NULL ) {
        fprintf( stderr, "Unable allocate resources\n" );
    }

    pci_detach( phdl );

    return EXIT_SUCCESS;
}

```

This code shows how the devnp-e1000.so driver determines whether or not the device supports MSI or MSI-X:

```

static int check_capabilities (i82544_dev_t *i82544, uint8_t bus,
                               uint8_t devfn)
{
    uint8_t    cap, cap_ptr = 0;
    uint16_t   device, dev_ctrl;
    int        offset;

    if (pci_read_config8 (bus, devfn, offsetof (struct _pci_config_regs,
        Capabilities_Pointer), 1, &cap_ptr) != PCI_SUCCESS)
        return (-1);
    if (pci_read_config16 (bus, devfn, offsetof (struct _pci_config_regs,
        Device_ID), 1, &device) != PCI_SUCCESS)
        return (-1);
    if (cap_ptr) {
        while (1) {
            if (pci_read_config8 (bus, devfn, cap_ptr, 1, &cap))
                return (-1);
            if (cap == PCI_CAP_MSI)
                i82544->msi_cap = 1;
            if (cap == PCI_CAP_MSI_X) {
                i82544->msi_cap = 2;
                i82544->msix_cap_ptr = cap_ptr;
            }
            if ((cap == PCI_CAP_PCI_EXPRESS) && (i82544->max_read != -1)) {
                offset = offsetof (struct _pci_capability_pci_express,
                    Device_Control);
                if (pci_read_config16 (bus, devfn, cap_ptr + offset, 1,
                    &dev_ctrl) != PCI_SUCCESS)
                    return (-1);
                dev_ctrl &= 0x8fff;
                dev_ctrl |= (i82544->max_read << 12);
                pci_write_config16 (bus, devfn, cap_ptr + offset, 1, &dev_ctrl);
            }
            if (pci_read_config8 (bus, devfn, cap_ptr + 1, 1, &cap_ptr))
                return (-1);
            if (! cap_ptr) {
                break;
            }
        }
    }
    return (EOK);
}

/* ... */

flags = PCI_INIT_ALL | PCI_INIT_IRQ | PCI_MASTER_ENABLE;
if (i82544->msi_cap)
    flags |= (i82544->msi_cap == 1) ? PCI_USE_MSI : PCI_USE_MSIX;

if ((i82544->pci_dev_hdl = pci_attach_device (NULL, flags, busindex,
    &info)) == NULL) {
    slogf (_SLOGC_NETWORK, _SLOG_ERROR, "pci_attach_device failed");
    err = errno;
    i82544_pci_detach_cleanup (i82544, 3);
    return (err);
}

```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_detach()

Disconnect from the PCI server

Synopsis:

```
#include <hw/pci.h>

int pci_detach( unsigned handle );
```

Arguments:

handle

The value returned by a successful call to [pci_attach\(\)](#) (p. 1964).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pci_detach()* function disconnects from the PCI server. Any resources allocated with [pci_attach_device\(\)](#) (p. 1966) are released.

The *pci_attach()* function opens a file descriptor against the PCI server, and all of the low-level library calls to the PCI server use this fd. When you call *pci_detach()*, the low-level code does a *close()* on the file descriptor, which tells the PCI server to clean up any allocations associated with it.



Don't call any of the other *pci_*()* functions after calling *pci_detach()* (unless you've reattached with *pci_attach()*).

Returns:

PCI_SUCCESS.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_detach_device()

Detach a driver from a PCI device

Synopsis:

```
#include <hw/pci.h>

int pci_detach_device( void* handle );
```

Arguments:

handle

The handle returned by [pci_attach_device\(\)](#) (p. 1966).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_detach_device()` function detaches a driver from a PCI device. Any resources allocated with `pci_attach_device()` are released, unless you attached the device with the `PCI_PERSIST` flag set.



You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.

Returns:

PCI_DEVICE_NOT_FOUND

No device could be found for *handle*.

PCI_SUCCESS

Success.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_find_class()

Find devices that have a specific class code

Synopsis:

```
#include <hw/pci.h>

int pci_find_class( unsigned long class_code,
                   unsigned index,
                   unsigned* bus,
                   unsigned* dev_func );
```

Arguments:

class_code

The class of device or function that you want to find. For a list of class codes, see <hw/pci.h>. You must OR together a class code and a subclass code (e.g. PCI_CLASS_DISPLAY | PCI_SUBCLASS_DISPLAY_XGA).

index

The index of the device or function that you want to find: 0 for the first, 1 for the second, and so on.

bus

The bus number, in the range [0...255].

dev_func

The device or function number of the *n*th device or function of the given class. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pci_find_class()* function determines the location of the *n*th PCI device or function that has the specified class code.



You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.

You can find all the devices having the same class code by making successive calls to this function, starting with an index of 0, and incrementing it until `PCI_DEVICE_NOT_FOUND` is returned.

Returns:**PCI_DEVICE_NOT_FOUND**

The device or function wasn't found.

PCI_SUCCESS

The device or function was found.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_find_device()

Find the PCI device with a given device ID and vendor ID

Synopsis:

```
#include <hw/pci.h>

int pci_find_device( unsigned device,
                    unsigned vendor,
                    unsigned index,
                    unsigned* bus,
                    unsigned* dev_func );
```

Arguments:

device

The device ID. For a list of supported device IDs, see `<hw/pci_devices.h>`.

vendor

The vendor ID. For a list of supported vendor IDs, see `<hw/pci_devices.h>`.

index

The index (*n*) of the device or function sought.

bus

A pointer to a location where the function can store the bus number of the device or function found.

dev_func

A pointer to a location where the function can store the device or function ID of the *n*th device or function found with the specified device and vendor IDs. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_find_device()` function returns the location of the *n*th PCI device that has the specified device and vendor IDs.



You must successfully call `pci_attach()` (p. 1964) before calling any of the other PCI functions.

You can find all the devices having the same device and vendor IDs by making successive calls to this function, starting with an index of 0, and incrementing it until `PCI_DEVICE_NOT_FOUND` is returned.

Returns:**PCI_DEVICE_NOT_FOUND**

The device or function wasn't found.

PCI_SUCCESS

The device or function was found.

-1

You haven't called `pci_attach()` (p. 1964), or the call to it failed.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_irq_routing_options()

Retrieve PCI IRQ routing information

Synopsis:

```
#include <hw/pci.h>

int pci_irq_routing_options(
    IRQRoutingOptionsBuffer * buf,
    uint32_t * irq );
```

Arguments:

buf

A pointer to a `IRQRoutingOptionsBuffer` structure where the function can store the IRQ routing information. For information about the layout of this buffer, see *PCI BIOS SPECIFICATION* Revision 2.1. You can get it from the PCI Special Interest Group at <http://pcisig.com/>.

irq

A pointer to a location where the function can store the current state of interrupts.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_irq_routing_options()` function returns the following:

- PCI interrupt routing options available on the system motherboard
- the current state of interrupts that are currently exclusively assigned to PCI.

Routing information is returned in a data buffer that contains an IRQ routing for each PCI device or slot.



You must successfully call `pci_attach()` (p. 1964) before calling any of the other PCI functions. The `pci_irq_routing_options()` function is for x86 only.

Returns:

PCI_SUCCESS

Success.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <inttypes.h>
#include <hw/pci.h>
#include <sys/neutrino.h>

struct {
    IRQRoutingOptionsBuffer buf;
    uint8_t databuf [2048];
} route_buf;

int main (void)
{
    int phdl;
    uint32_t irq;

    if ((phdl = pci_attach (0)) == -1) {
        printf ("Unable to attach - errno %s\n", strerror (errno));
        exit (1);
    }

    memset (route_buf.databuf, 0, sizeof (route_buf.databuf));
    route_buf.buf.BufferSize = sizeof (route_buf.databuf);
    if (pci_irq_routing_options (&route_buf.buf, &irq) !=
        PCI_SUCCESS) {
        printf ("Routing option failed - errno %s\n",
            strerror (errno));
        exit (1);
    }

    printf ("PCI Irq Map = %x\n", irq);
    pci_detach (phdl);
    return (0);
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_map_irq()

Map an interrupt pin to an IRQ

Synopsis:

```
#include <hw/pci.h>

int pci_map_irq( unsigned bus,
                 unsigned dev_func,
                 short intno,
                 short intpin );
```

Arguments:

bus

The bus number of the device.

dev_func

The device or function number of the device. The device number is in bits 7 through 3, and the function number is in bits 2 through 0.

intno

The interrupt to be mapped (e.g. 0 - 15 on x86).

intpin

The PCI interrupt pin (0x0a - 0x0d).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pci_map_irq()* function maps a PCI interrupt pin to a specific interrupt request (IRQ).



You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.

Returns:

PCI_SUCCESS

Success.

PCI_SET_FAILED

The PCI server was unable to map the *intno/intpin*.

PCI_UNSUPPORTED_FUNCTION

This function isn't supported.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_present()

Determine whether or not PCI BIOS is present

Synopsis:

```
#include <hw/pci.h>

int pci_present( unsigned* lastbus,
                 unsigned* version,
                 unsigned* hardware );
```

Arguments:

lastbus

The number of the last PCI bus in the system. PCI buses are numbered from 0, up to and including this value.

version

The version number of the PCI interface.

hardware

The specific hardware characteristics the platform supports with regard to accessing configuration space and generating PCI Special Cycles.

The PCI specification defines two hardware mechanisms for accessing configuration space. Bit 0 of *hardware* is set (1) if mechanism 1 is supported, and reset (0) otherwise. Bit 1 is set (1) if mechanism 2 is supported, and reset (0) otherwise.

The specification also defines hardware mechanisms for generating Special Cycles. Bit 4 of *hardware* is set (1) if the platform supports Special Cycle generation based on Config Mechanism 1, and reset (0) otherwise. Bit 5 is set (1) if the platform supports Special Cycle generation based on Config Mechanism 2, and reset (0) otherwise.

The arguments can be NULL if you just want to check for PCI capabilities.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pci_present()* function determines whether or not the PCI BIOS interface function set is present. It also determines the following:

- the current interface version
- what hardware mechanism for accessing configuration space is supported
- whether or not the hardware supports the generation of PCI Special Cycles.



You must successfully call *pci_attach()* (p. 1964) before calling any of the other PCI functions.

Returns:

-1

PCI BIOS isn't present.

PCI_SUCCESS

PCI BIOS is present.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_read_config()

Read from the configuration space of a PCI device

Synopsis:

```
#include <hw/pci.h>

int pci_read_config( void* handle,
                    unsigned offset,
                    unsigned count,
                    size_t size,
                    void* buff );
```

Arguments:

handle

The handle returned by [pci_attach_device\(\)](#) (p. 1966).

offset

The offset into the configuration space where you want to read from.

count

The number of objects that you want to read.

size

The size of each object.

buff

A pointer to a buffer where the function can store the objects that it reads.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_read_config()* function reads *count* objects of the specified *size* into *buff* at the given *offset* from the configuration space of the PCI device specified by *handle*.



- You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.
- The [pci_read_config*\(\)](#) functions can return inconsistent data if another process is extensively using the PCI bus.

Returns:**PCI_BAD_REGISTER_NUMBER**

The offset is invalid.

PCI_BUFFER_TOO_SMALL

The PCI BIOS server reads only 100 bytes at a time; *size* is too large.

PCI_DEVICE_NOT_FOUND

The *handle* is invalid.

PCI_SUCCESS

Success.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_read_config8()

Read a byte from the configuration space of a device

Synopsis:

```
#include <hw/pci.h>

int pci_read_config8( unsigned bus,
                     unsigned dev_func,
                     unsigned offset,
                     unsigned count,
                     char* buff );
```

Arguments:

bus

The bus number.

dev_func

The name of the device or function.

offset

The register offset into the configuration space, in the range [0...255].

count

The number of bytes to read.

buff

A pointer to a buffer where the requested bytes are placed.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_read_config8()* function reads the specified number of bytes from the configuration space of the given device or function.



- You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.
- The [pci_read_config*\(\)](#) functions can return inconsistent data if another process is extensively using the PCI bus.

Returns:**PCI_BAD_REGISTER_NUMBER**

An invalid register offset was given.

PCI_BUFFER_TOO_SMALL

The PCI BIOS server reads only 100 bytes at a time; *count* is too large.

PCI_SUCCESS

The device or function was found.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Examples:

See [pci_attach_device\(\)](#) (p. 1966).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_read_config16()

Read 16-bit values from the configuration space of a device

Synopsis:

```
#include <hw/pci.h>

int pci_read_config16( unsigned bus,
                      unsigned dev_func,
                      unsigned offset,
                      unsigned count,
                      char* buff );
```

Arguments:

bus

The bus number.

dev_func

The name of the device or function.

offset

The register offset into the configuration space. This offset must be aligned to a 16-bit boundary (that is 0, 2, 4, ..., 254 bytes).

count

The number of 16-bit values to read.

buff

A pointer to a buffer where the requested 16-bit values are placed.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_read_config16()* function reads the specified number of 16-bit values from the configuration space of the given device or function.



- You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.
- The [pci_read_config*\(\)](#) functions can return inconsistent data if another process is extensively using the PCI bus.

Returns:**PCI_BAD_REGISTER_NUMBER**

An invalid offset register number was given.

PCI_BUFFER_TOO_SMALL

The PCI BIOS server reads only 50 words at a time; *count* is too large.

PCI_SUCCESS

The device or function was found.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Examples:

See [pci_attach_device\(\)](#) (p. 1966).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_read_config32()

Read 32-bit values from the configuration space of a device

Synopsis:

```
#include <hw/pci.h>

int pci_read_config32( unsigned bus,
                      unsigned dev_func,
                      unsigned offset,
                      unsigned count,
                      char* buff );
```

Arguments:

bus

The bus number.

dev_func

The name of the device or function.

offset

The register offset into the configuration space. This offset must be aligned to a 32-bit boundary (that is 0, 4, 8, ..., 252 bytes).

count

The number of 32-bit values to read.

buff

A pointer to a buffer where the requested 32-bit values are placed.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_read_config32()* function reads the specified number of 32-bit values from the configuration space of the given device or function.



- You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.
- The [pci_read_config*\(\)](#) functions can return inconsistent data if another process is extensively using the PCI bus.

Returns:**PCI_BAD_REGISTER_NUMBER**

An invalid register offset was given.

PCI_SUCCESS

The device or function was found.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_rescan_bus()

Rescan the PCI bus for added or removed devices

Synopsis:

```
#include <hw/pci.h>

int pci_rescan_bus( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_rescan_bus()* function asks the PCI server to rescan the PCI bus(es) for devices that have been inserted or removed. This is used in hot swap situations such as for CardBus cards. The PCI server updates its internal configuration to reflect any changes.



You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.

Returns:

PCI_SUCCESS

Success.

-1

The function failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

pci_write_config()

Write to the configuration space of a PCI device

Synopsis:

```
#include <hw/pci.h>

int pci_write_config( void* handle,
                     unsigned offset,
                     unsigned count,
                     size_t size,
                     const void* buff );
```

Arguments:

handle

The handle returned by [pci_attach_device\(\)](#) (p. 1966).

offset

The offset into the configuration space where you want to write the data.

count

The number of objects that you want to write.

size

The size of each object.

buff

A pointer to the data that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_write_config()` function writes *count* objects of the specified *size* from *buff* at the given *offset* to the configuration space of the PCI device specified by *handle*.



You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.

Returns:

PCI_BAD_REGISTER_NUMBER

The *offset* specified is invalid.

PCI_BUFFER_TOO_SMALL

The *size* argument is too large.

PCI_SET_FAILED

An error occurred writing to the configuration space of the device.

PCI_SUCCESS

Success.

PCI_UNSUPPORTED_FUNCT

This device doesn't support writing to its configuration space.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_write_config8()

Write bytes to the configuration space of a PCI device

Synopsis:

```
#include <hw/pci.h>

int pci_write_config8( unsigned bus,
                      unsigned dev_func,
                      unsigned offset,
                      unsigned count,
                      char* buff );
```

Arguments:

bus

The bus number.

dev_func

The device or function ID. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

offset

The register offset into the configuration space, in the range [0...255].

count

The number of bytes to write.

buff

A pointer to a buffer containing the data to be written into the configuration space.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pci_write_config8()* function writes individual bytes to the configuration space of the specified device.



You must successfully call [pci_attach\(\)](#) (p. 1964) before calling any of the other PCI functions.

Returns:

PCI_BAD_REGISTER_NUMBER

An invalid offset register number was given.

PCI_BUFFER_TOO_SMALL

The *size* argument is greater than 100 bytes.

PCI_SUCCESS

The device or function was found.

-1

You haven't called [pci_attach\(\)](#) (p. 1964), or the call to it failed.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_write_config16()

Write 16-bit values to the configuration space of a device

Synopsis:

```
#include <hw/pci.h>

int pci_write_config16( unsigned bus,
                        unsigned dev_func,
                        unsigned offset,
                        unsigned count,
                        char* buff );
```

Arguments:

bus

The bus number.

dev_func

The device or function ID. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

offset

The offset into the configuration space. This must be aligned to a 16-bit boundary (that is 0, 2, 4, ..., 254 bytes).

count

The number of 16-bit values to write.

buff

A pointer to a buffer containing the data to be written into the configuration space.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_write_config16()` function writes individual 16-bit values to the configuration space of the specified device.



You must successfully call `pci_attach()` (p. 1964) before calling any of the other PCI functions.

Returns:**PCI_BAD_REGISTER_NUMBER**

An invalid register offset was given.

PCI_BUFFER_TOO_SMALL

The *size* argument is greater than 50 words.

PCI_SUCCESS

The device or function was found.

-1

You haven't called `pci_attach()` (p. 1964), or the call to it failed.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pci_write_config32()

Write 32-bit values to the configuration space of a device

Synopsis:

```
#include <hw/pci.h>

int pci_write_config32( unsigned bus,
                        unsigned dev_func,
                        unsigned offset,
                        unsigned count,
                        char* buff );
```

Arguments:

bus

The bus number.

dev_func

The device or function ID. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

offset

The register offset into the configuration space. This must be aligned to a 32-bit boundary (that is 0, 4, 8, ..., 252 bytes).

count

The number of 32-bit values to write.

buff

A pointer to a buffer containing the data to be written into the configuration space.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_write_config32()` function writes individual 32-bit values to the configuration space of the specified device.



You must successfully call `pci_attach()` (p. 1964) before calling any of the other PCI functions.

Returns:**PCI_BAD_REGISTER_NUMBER**

An invalid register offset was given.

PCI_SUCCESS

The device or function was found.

-1

You haven't called `pci_attach()` (p. 1964), or the call to it failed.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pclose()

Close a pipe

Synopsis:

```
#include <stdio.h>

int pclose( FILE* stream );
```

Arguments:

stream

The stream pointer for the pipe that you want to close, that you obtained by calling [popen\(\)](#) (p. 2019).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `pclose()` function closes the pipe associated with *stream*, and waits for the subprocess created by [popen\(\)](#) to terminate.

Returns:

The termination status of the command language interpreter, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EINTR

The `pclose()` function was interrupted by a signal while waiting for the child process to terminate.

ECHILD

The *pclose()* function was unable to obtain the termination status of the child process.

Examples:

See [popen\(\)](#) (p. 2019).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

perror()

Print an error message associated with `errno`

Synopsis:

```
#include <stdio.h>

void perror( const char *prefix );
```

Arguments:

prefix

NULL, or a string that you want to print before the error message.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `perror()` function prints the following to [stderr](#) (p. 3175):

- the given *prefix*, followed by “: ”
- the error message returned by [strerror\(\)](#) (p. 3198) for the current value of [errno](#) (p. 539)
- a newline character.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;

    fp = fopen( "data.fil", "r" );
    if( fp == NULL ) {
        perror( "Unable to open file" );
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

pipe()

Create a pipe

Synopsis:

```
#include <unistd.h>

int pipe( int fildes[2] );
```

Arguments:

fildes

An array where the function can store the file descriptors for the ends of the pipe.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pipe()* function creates a pipe (an unnamed FIFO) and places a file descriptor for the read end of the pipe in *fildes*[0], and a file descriptor for the write end of the pipe in *fildes*[1]. Their integer values are the two lowest available at the time of the *pipe()* function call. The `O_NONBLOCK` flag is cleared for both file descriptors. (You can use [fcntl\(\)](#) (p. 623) to set the `O_NONBLOCK` flag.)

You can write data to file descriptor *fildes*[1] and read it from file descriptor *fildes*[0]. If you read from file descriptor *fildes*[0], it returns the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

The pipe buffer is allocated by the `pipe` resource manager.

You typically use this function to connect standard utilities acting as filters, passing the write end of the pipe to the data-producing process as its `STDOUT_FILENO`, and the read end of the pipe to the data-consuming process as its `STDIN_FILENO` (either via the traditional [fork\(\)](#) (p. 685), [dup2\(\)](#) (p. 485), or `exec*`, or the more efficient *spawn** calls).

If successful, *pipe()* marks the *st_mtime*, *st_ctime*, *st_atime* and *st_mtime* fields of the pipe for updating.

Returns:**0**

Success.

-1An error occurred ([errno](#) (p. 539) is set).**Errors:****EMFILE**

The calling process doesn't have at least 2 unused file descriptors available.

ENFILE

The number of simultaneously open files in the system would exceed the configured limit.

ENOSPC

There's insufficient space available to allocate the pipe buffer.

ENOSYS

There's no pipe manager running.

EROFS

The pipe pathname space is a read-only filesystem.

Classification:[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

poll()

Multiplex input/output over a set of file descriptors

Synopsis:

```
#include <sys/poll.h>

int poll( struct pollfd *fds,
          nfds_t nfds,
          int timeout );
```

Arguments:

fds

An array of pollfd structures that defines the file descriptors of interest.

nfds

The number of elements in the *fds* array.

timeout

The timeout, in milliseconds.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *poll()* function provides applications with a mechanism for multiplexing input/output over a set of file descriptors.



Polling can interfere with the kernel's efforts to manage power usage. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

The *fds* argument is an array of pollfd structures:

```
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

The members are:

fd

The file descriptor to be examined.

events

Flags that indicate the type of events to look for.

revents

Returned event flags.

For each member of the array pointed to by *fds*, *poll()* examines the given file descriptor for the event(s) specified in *events*. The number of `pollfd` structures in the *fds* array is specified by *nfds*. The array's members are `pollfd` structures within which *fd* specifies an open file descriptor, and *events* and *revents* are bitmasks constructed by ORing a combination of the following event flags:

POLLERR

An error has occurred on the device. This flag is valid only in the *revents* bitmask; it's ignored in the *events* member.

POLLHUP

The device has been disconnected. This event and `POLLOUT` are mutually exclusive; a device can never be writable if a hangup has occurred. However, this event and `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, or `POLLPRI` are not mutually exclusive.



If the remote end of a socket is closed, *poll()* indicates both `POLLIN` and `POLLHUP` events.

This flag is valid only in the *revents* bitmask; it's ignored in the *events* member.

POLLIN

Data other than high-priority data may be read without blocking. This is equivalent to `POLLRDNORM` | `POLLRDBAND`.

POLLNVAL

The specified *fd* value is invalid. This flag is valid only in the *revents* member; it's ignored in the *events* member.

POLLOUT

Normal data may be written without blocking.

POLLPRI

High-priority data may be read without blocking.

POLLRDBAND

Priority data may be read without blocking.

POLLRDNORM

Normal data may be read without blocking.

POLLWRBAND

Priority data may be written.

POLLWRNORM

Equivalent to `POLLOUT`.

The significance and semantics of normal, priority, and high-priority data are file- and device-specific.

If the value of *fd* is less than 0, *events* is ignored, and *revents* is set to 0 in that entry on return from *poll()*.

In each `pollfd` structure, *poll()* clears the *revents* member, except that where the application requested a report on a condition by setting one of the bits of *events* listed above, *poll()* sets the corresponding bit in *revents* if the requested condition is true. In addition, *poll()* sets the `POLLHUP`, `POLLERR`, and `POLLNVAL` flag in *revents* if the condition is true, even if the application didn't set the corresponding bit in *events*.

If none of the defined events occurs on any selected file descriptor, *poll()* waits at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value of *timeout* is 0, *poll()* returns immediately. If the value of *timeout* is -1, *poll()* blocks until a requested event occurs or until the call is interrupted.

The *poll()* function isn't affected by the `O_NONBLOCK` flag.

The *poll()* function reports regular files, terminal and pseudo-terminal devices, FIFOs, and pipes.

Regular files always poll TRUE for reading and writing.

A file descriptor for a socket that's listening for connections indicates that it's ready for reading, once connections are available. A file descriptor for a socket that connects asynchronously indicates that it's ready for writing, once a connection has been established.

See also `<sys/poll.h>`.

Returns:

> 0

The total number of file descriptors that have been selected.

0

The call timed out, and no file descriptor has been selected.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The allocation of internal data structures failed, but a subsequent request may succeed.

EINTR

A signal was caught during *poll()*.

EFAULT

The *fds* argument pointed to a nonexistent portion of the calling process's address space.

Examples:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/poll.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

struct sockaddr_in sad;

void *
client(void *arg)
{
    int s;
    const char *p = "Some data\n";

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return NULL;
    }

    if (connect(s, (struct sockaddr *)&sad, sizeof(sad)) == -1) {
        perror("connect");
        return NULL;
    }

    write(s, p, strlen(p));
    close(s);

    return NULL;
}
```

```

int
main(void)
{
    struct pollfd fds;
    int s = -1, s2 = -1, done_accept = 0, oflags, ret;
    char buf[100];

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return 1;
    }

    sad.sin_family      = AF_INET;
    sad.sin_len         = sizeof(sad);
    sad.sin_addr.s_addr = inet_addr("127.0.0.1");
    sad.sin_port        = htons(1234);

    fds.fd = s;
    fds.events = POLLRDNORM;

    oflags = fcntl(s, F_GETFL);
    oflags |= O_NONBLOCK;
    fcntl(s, F_SETFL, oflags);

    if (bind(s, (struct sockaddr *)&sad, sizeof(sad)) == -1) {
        perror("bind");
        return 1;
    }

    listen(s, 5);

    if ((ret = pthread_create(NULL, NULL, client, NULL)) != EOK) {
        fprintf(stderr, "pthread_create: %s\n", strerror(ret));
        return 1;
    }

    for (;;) {
        if ((ret = poll(&fds, 1, -1)) == -1) {
            perror("poll");
            break;
        }

        else if (ret != 1 || (fds.revents & POLLRDNORM) == 0) {
            break;
        }

        if (done_accept) {
            if ((ret = read(s2, buf, sizeof(buf))) <= 0) {
                break;
            }

            printf("%s", buf);
        }
        else {
            if ((s2 = accept(s, NULL, 0)) == -1) {
                perror("accept");
                break;
            }

            fds.fd = s2;
            done_accept = 1;
        }
    }

    close(s);
    close(s2);

    return 0;
}

```

Classification:*POSIX 1003.1 XSI* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Not all managers support POLLPRI, POLLWRBAND, POLLERR, and POLLHUP.

popen()

Execute a command, creating a pipe to it

Synopsis:

```
#include <stdio.h>

FILE* popen( const char* command,
             const char* mode );
```

Arguments:

command

The command that you want to execute.

mode

The I/O mode for the pipe, which must be "r" or "w"; see below.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *popen()* function executes the command specified by *command* and creates a pipe between the calling process and the executed command.

Depending on the *mode* argument, you can use the returned stream pointer to read from or write to the pipe.

The executed command has the same environment as its parents. The command is started as follows:

```
spawnlp (P_NOWAIT, shell_command, shell_command,
         "-c", command, (char*) NULL );
```

where *shell_command* is the command specified by the **SHELL** environment variable (if it exists), or the sh utility.

The *mode* argument to *popen()* is a string that specifies an I/O mode for the pipe:

- If *mode* is "r", then when the child process is started:
 - Its file descriptor, `STDOUT_FILENO`, is the writable end of the pipe.
 - The *fileno(stream)* in the calling process is the readable end of the pipe, where *stream* is the stream pointer returned by *popen()*.
- If *mode* is "w", then when the child process is started:
 - Its file descriptor, `STDIN_FILENO`, is the readable end of the pipe.
 - The *fileno(stream)* in the calling process is the writable end of the pipe, where *stream* is the stream pointer return by *popen()*.
- If *mode* is any other value, the result is undefined.



Use [pclose\(\)](#) (p. 2007) to close a stream that you used *popen()* to open.

Returns:

A non-NULL stream pointer on successful completion. If *popen()* is unable to create either the pipe or the subprocess, it returns a NULL stream pointer and sets [errno](#) (p. 539).

Errors:

EINVAL

The *mode* argument is invalid.

ENOSYS

There's no pipe manager running.

The *popen()* function may also set *errno* values as described by the [pipe\(\)](#) (p. 2011) and [spawnl\(\)](#) (p. 3099) functions.

Examples:

```
/*
 * upper:  executes a given program, converting all input
 *         to upper case.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <limits.h>

char  buffer[_POSIX_ARG_MAX];

int main( int argc, char** argv )
{
    int  i;
    int  c;
    FILE* f;
```



```

for( i = 1; i < argc; i++ ) {
    strcat( buffer, argv[i] );
    strcat( buffer, " " );
}
if( ( f = popen( buffer, "w" ) ) == NULL ) {
    perror( "popen" );
    return EXIT_FAILURE;
}
while( ( c = getchar() ) != EOF ) {
    if( islower( c ) )
        c = toupper( c );
    putc( c, f );
}
pclose( f );
return EXIT_SUCCESS;
}

```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

posix_fadvise(), posix_fadvise64()

Provide advisory information about the expected use of a file

Synopsis:

```
#include <fcntl.h>

int posix_fadvise( int fd,
                  off_t offset,
                  size_t len,
                  int advice );

int posix_fadvise64( int fd,
                    off64_t offset,
                    size_t len,
                    int advice );
```

Arguments:

fd

An open file descriptor.

offset

The offset of the data about which you want to provide advice.

len

The length of the data.

advice

The advice you want to give; one of:

- **POSIX_FADV_NORMAL** — the application has no advice to give on its behavior with respect to the specified data. This is the default characteristic if no advice is given for an open file.
- **POSIX_FADV_SEQUENTIAL** — the application expects to access the specified data sequentially from lower offsets to higher offsets.
- **POSIX_FADV_RANDOM** — the application expects to access the specified data in a random order.
- **POSIX_FADV_WILLNEED** — the application expects to access the specified data in the near future.
- **POSIX_FADV_DONTNEED** — the application expects that it will not access the specified data in the near future.

- **POSIX_FADV_NOREUSE** — the application expects to access the specified data once and then not reuse it thereafter.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_fadvise()* and *posix_fadvise64()* functions advise the filesystem how the application expects to use the data in the file associated with the open file descriptor, *fd*, starting at *offset* and continuing for *len* bytes. The *posix_fadvise64()* is a large-file version of *posix_fadvise()*.

The specified range need not currently exist in the file. If *len* is zero, all data following *offset* is specified. The filesystem may use this information to optimize handling of the specified data. The *posix_fadvise()* function has no effect on the semantics of other operations on the specified data, although it may affect the performance of other operations.

Returns:

0

Success.

EBADF

The *fd* argument isn't a valid file descriptor.

EINVAL

The value of *advice* is invalid.

ESPIPE

The *fd* argument is associated with a pipe or FIFO.

Classification:

posix_fadvise() is [POSIX 1003.1 ADV](#) (p. 46); *posix_fadvise64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_fallocate(), posix_fallocate64()

Allocate space for a file on the filesystem

Synopsis:

```
#include <fcntl.h>

int posix_fallocate( int fd,
                    off_t offset,
                    off_t len );

int posix_fallocate64( int fd,
                      off64_t offset,
                      off64_t len );
```

Arguments:

fd

A file descriptor for the file you want to allocate space for.

offset

The offset into the file where you want to allocate space.

len

The number of bytes to allocate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_fallocate()* and *posix_fallocate64()* functions ensure that any required storage for regular file data starting at *offset* and continuing for *len* bytes is allocated on the filesystem storage media. The *posix_fallocate64()* function is a large-file version of *posix_fallocate()*.

If *posix_fallocate()* returns successfully, subsequent writes to the specified file data won't fail due to the lack of free space on the filesystem storage media.

If the *offset + len* is beyond the current file size, then *posix_fallocate()* adjusts the file size to be *offset + len*. Otherwise, the file size isn't changed.

You can free the allocated space by using [creat\(\)](#) (p. 381) or [open\(\)](#) (p. 1905) to truncate the file, or by calling [ftruncate\(\)](#) (p. 793) to make the size of the file less than *offset + len*.

Returns:

EOK

Success.

EBADF

The *fd* argument isn't a valid file descriptor.

EBADF

The *fd* argument references a file that was opened without write permission.

EFBIG

The value of *offset + len* is greater than the maximum file size.

EINTR

A signal was caught during execution.

EINVAL

The *len* argument was zero or the *offset* argument was less than zero.

EIO

An I/O error occurred while reading from or writing to a filesystem.

ENODEV

The *fd* argument doesn't refer to a regular file.

ENOSPC

There's insufficient free space remaining on the filesystem storage media.

ESPIPE

The *fd* argument is associated with a pipe or FIFO.

Classification:

posix_fallocate() is [POSIX 1003.1 ADV](#) (p. 46); *posix_fallocate64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_madvise()

Provide advisory information about the expected use of memory

Synopsis:

```
#include <sys/mman.h>

int posix_madvise( void *addr,
                  size_t len,
                  int advice );
```

Arguments:

addr

The address of the data the application is providing advice about.

len

The length of the data, in bytes.

advice

The advice to be applied to the memory range; one of the following:

- **POSIX_MADV_NORMAL** — the application has no advice to give on its behavior with respect to the specified range. This is the default characteristic if no advice is given for a range of memory.
- **POSIX_MADV_SEQUENTIAL** — the application expects to access the data sequentially from lower addresses to higher addresses.
- **POSIX_MADV_RANDOM** — the application expects to access the data in a random order.
- **POSIX_MADV_WILLNEED** — the application expects to access the data in the near future.
- **POSIX_MADV_DONTNEED** — the application expects that it won't access the data in the near future.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_madvise()* function advises the memory manager how the application expects to use the data in the memory starting at address *addr*, and continuing for *len* bytes. The memory manager may use this information to optimize handling of the data.

Returns:**EOK**

Success.

EINVAL

The value of *advice* is invalid.

ENOMEM

Addresses in the specified range are partly or completely outside the range allowed for the address space of the calling process.

Classification:

[POSIX 1003.1 ADV](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_mem_offset(), posix_mem_offset64()

Get the offset and length of a mapped memory block

Synopsis:

```
#include <sys/mman.h>

int posix_mem_offset( const void *addr,
                    size_t len,
                    off_t *off,
                    size_t *contig_len,
                    int *fildes );

int posix_mem_offset64( const void *addr,
                      size_t len,
                      off64_t *off,
                      size_t *contig_len,
                      int *fildes );
```

Arguments:

addr

The address that the memory block is currently mapped at.

len

The length of the block of memory object that you want the offset of.

off

A pointer to a location where the function can store the offset (or location) within the memory block.

contig_len

A pointer to a location where the function can store either the value of *len* or the length of the largest contiguous block of the memory object that's currently mapped to the calling process starting at *addr*, whichever is smaller.

fildes

A pointer to a location where the function can store the file descriptor used to establish the mapping that contains *addr*.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *posix_mem_offset()* and *posix_mem_offset64()* functions set the variable pointed to by *off* to the offset (or location), within a memory object, of the memory block currently mapped at *addr*. The *posix_mem_offset64()* function is a 64-bit version of *posix_mem_offset()*.

The *posix_mem_offset()* function sets the variable pointed to by *fildes* to the descriptor used (via *mmap()* (p. 1656)) to establish the mapping that contains *addr*, or -1 if the descriptor has since been closed.

This function supports typed memory, shared memory, and memory-mapped files, but not anonymous memory.

The *len* argument specifies the length of the block of memory you want the offset for. On return, the value pointed to by *contig_len* is either *len* or the length of the largest contiguous block of the memory object that's currently mapped to the calling process starting at *addr*, whichever is smaller.

If the memory object mapped at *addr* is a typed memory object, then if you use the *off* and *contig_len* values that you obtained by calling *posix_mem_offset()* in a call to *mmap()* with a file descriptor that refers to the same memory pool as *fildes* (either through the same port or through a different port), and that was opened with neither the `POSIX_TYPED_MEM_ALLOCATE` nor the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the typed memory area that is mapped is exactly the same area that was mapped at *addr* in the address space of the process that called *posix_mem_offset()*.

Returns:

0

Success.

EACCES

The process hasn't mapped a memory object at the given address *addr*.

EOVERFLOW

This error is specific to *posix_mem_offset()* and is returned when the address is too large for the 32-bit `off_t`; use *posix_mem_offset64()* instead.

Classification:

posix_mem_offset() is [POSIX 1003.1 TYM](#); (p. 46) *posix_mem_offset64()* is for [large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_memalign()

Allocate aligned memory

Synopsis:

```
#include <stdlib.h>

int posix_memalign( void ** memptr,
                   size_t alignment,
                   size_t size );
```

Arguments:

memptr

A pointer to a location where *posix_memalign()* can store a pointer to the memory.

alignment

The alignment to use for the memory. This must be a multiple of `size( void * )` that's also a power of 2.

size

The size, in bytes, of the block to allocate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_memalign()* function allocates *size* bytes aligned on a boundary specified by *alignment*. It returns a pointer to the allocated memory in *memptr*.

The buffer allocated by *posix_memalign()* is contiguous in virtual address space, but not physical memory. Since some platforms don't allocate memory in 4 KB page sizes, you shouldn't assume that the memory allocated will be physically contiguous if you specify a size of 4 KB or less.

You can obtain the physical address of the start of the buffer using [mem_offset\(\)](#) (p. 1600) with *fd=NOFD*.

Returns:**0**

Success.

-1An error occurred ([errno](#) (p. 539) is set).**Errors:****EINVAL**

The value of *alignment* isn't a multiple of `size( void * )` that's also a power of 2, or it's greater than the value of the **MALLOC_MAX_ALIGNMENT** environment variable.

ENOMEM

There's insufficient memory available with the requested alignment.

Classification:[POSIX 1003.1 ADV](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_openpt()

Open a pseudo-terminal device

Synopsis:

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt( int oflag );
```

Arguments:

oflag

Flags that specify the file status flags and file access modes of the open file description; a combination of the following:

- **O_RDWR** — open for reading and writing.
- **O_NOCTTY** — don't make the terminal device become the controlling terminal for the process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `posix_openpt()` function establishes a connection between a master device for a pseudo-terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo-terminal.

Returns:

A file descriptor for the master pseudo-terminal, or `-1` if an error occurred ([errno](#) (p. 539) is set).

Errors:

EMFILE

OPEN_MAX file descriptors are currently open in the calling process.

ENFILE

The maximum allowable number of files is currently open in the system.

EINVAL

The value of *oflag* isn't valid.

EAGAIN

Out of pseudo-terminal resources.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_spawn(), posix_spawnp()

Spawn a process

Synopsis:

```
#include <spawn.h>

int posix_spawn( pid_t *_Restrict pid,
                 const char *_Restrict path,
                 const posix_spawn_file_actions_t *
                 file_actions,
                 const posix_spawnattr_t *_Restrict attrp,
                 char * const argv[_Restrict_arr],
                 char * const envp[_Restrict_arr] );

int posix_spawnp( pid_t *_Restrict pid,
                  const char *_Restrict file,
                  const posix_spawn_file_actions_t *
                  file_actions,
                  const posix_spawnattr_t *_Restrict attrp,
                  char * const argv[_Restrict_arr],
                  char * const envp[_Restrict_arr] );
```

Arguments:

pid

NULL, or a pointer to a location where the function can store the process ID of the child process.

path

(*posix_spawn()* only) An absolute path to the executable. Use *posix_spawnp()* to search for the executable to spawn.

file

(*posix_spawnp()* only) The name of the executable file, which is used to construct a pathname that identifies the new process image file. If the *file* parameter contains a slash character, the file parameter is used as the pathname for the new process image file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable **PATH**. If this environment variable isn't defined, the results of the search are implementation-defined.

file_actions

NULL, or a pointer to an opaque spawn file actions object that specifies the actions you want to perform on the file descriptors in the child process. For more information, see “File actions,” below.

attrp

NULL, or a pointer to an opaque spawn attributes object that describes attributes to be applied during the spawn. If *attrp* is NULL, then the default attribute values are used. For more information, see the “Attributes” section, below.

argv

A pointer to an argument vector. The value in *argv[0]* should represent the filename of the program being loaded, but can be NULL if no arguments are being passed. The last member of *argv* *must* be NULL.

As a QNX Neutrino extension, the value of *argv* can be NULL. In this case, the behavior is exactly as if *argv* were a pointer to the two-element array:

- {*path*, NULL} for *posix_spawn()*
- {*file*, NULL} for *posix_spawnp()*

envp

A pointer to an array of character pointers, each pointing to a string defining an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable. If the value of *envp* is NULL, then the child process inherits the environment of the parent process.

If a variable appears more than once in the *envp* array, the behavior is undefined.



The `__Restrict_arr` attribute informs the compiler that the *argv* and *envp* pointers don't alias each other, and that the compiler is therefore free to perform certain optimizations that otherwise might not be legal.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawn()` and `posix_spawnp()` functions create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the *new process image file*. The difference between these two functions is only in the means by which they specify the new process image file:

- `posix_spawn()` takes an absolute path to the executable
- `posix_spawnp()` performs a path search for the file

A running process has two sets of credentials of importance during its execution: the real user and group IDs (*ruid* and *rgid* respectively) and the effective user and group IDs (*euid* and *egid* respectively). Typically, when a new process (a child process) is spawned, both sets of credentials (real and effective) are taken from the spawning (parent) process. The exception to this is if the executable file being spawned has either its *setuid* and/or *setgid* mode bits set, in which case the *euid* is set to the *uid* of the executable, and/or the *egid* is set to the *gid* of the executable. POSIX doesn't define a way to directly specify the *ruid* and *rgid* of a spawned process. If you want to do this, call `posix_spawnattr_setcred()` (p. 2095) —a QNX Neutrino extension—before spawning the child process.

The ability to do certain operations is controlled by `procmgr_ability()` (p. 2141):

The ability to:	Is controlled by:
Spawn new processes	PROC_MGR_AID_SPAWN
Set the group ID of the child process	PROC_MGR_AID_SPAWN_SETGID
Set the user ID of the child process	PROC_MGR_AID_SPAWN_SETUID
Create a new application ID for the child process	PROC_MGR_AID_CHILD_NEWAPP

When a C program is executed as the result of calling `posix_spawn()` or `posix_spawnp()`, it's entered as a C-language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves.

In addition, the global *environ* variable is initialized as a pointer to an array of character pointers to the environment strings. The argument *argv* is an array of character pointers to NULL-terminated strings. The last member of this array is a NULL pointer and isn't counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv[0]* should point to a filename that's associated with the process image that you're starting.

The number of bytes available for the child process's combined argument and environment lists is **{ARG_MAX}**.

On successful completion, *posix_spawn()* and *posix_spawnnp()* store the process ID of the child process in the variable pointed to by a non-NULL *pid* argument, and return zero as the function return value. Otherwise, no child process is created, the value stored in the variable pointed to by a non-NULL *pid* is unspecified, and the functions return an error number. If the *pid* argument is NULL, the process ID of the child isn't returned to the caller.

File actions

The spawn file actions let you control which files are open in the spawned process. These actions are defined in an opaque *posix_spawn_file_actions_t* object.

If *file_actions* is NULL, then the file descriptors that are open in the calling process remain open in the child process, except for those whose close-on-exec flag, *FD_CLOEXEC*, is set. For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks, remain unchanged. For more information about *FD_CLOEXEC* and file locks, see [fcntl\(\)](#) (p. 623).

If *file_actions* isn't NULL, then the file descriptors open in the child process are those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the *FD_CLOEXEC* flag for each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions is:

1. The set of open file descriptors for the child process is initially the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks, remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process are changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object are performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its *FD_CLOEXEC* flag set is closed.

To initialize a file actions object, call [posix_spawn_file_actions_init\(\)](#) (p. 2057), and then:

Call:	To specify a file descriptor that you want to:
posix_spawn_file_actions_addclose() (p. 2048)	Close
posix_spawn_file_actions_adddup2() (p. 2050)	Duplicate with dup2() (p. 485)
posix_spawn_file_actions_addopen() (p. 2052)	Open

To destroy the file actions object, call [*posix_spawn_file_actions_destroy\(\)*](#) (p. 2055).

Attributes

The spawn attributes are defined in an opaque `posix_spawnattr_t` object. Use [*posix_spawnattr_init\(\)*](#) (p. 2093) to initialize the object, and [*posix_spawnattr_destroy\(\)*](#) (p. 2064) to destroy it.



In order to set an attribute, you must typically call the appropriate *posix_spawnattr_**(*)* function *and* set the corresponding flag.

By default (i.e. if you don't provide a spawn attributes object, or you don't set the appropriate attribute):

- The child process inherits the parent's process group.
- Signals set to the default action in the parent process are set to the default action in the child process.
- Signals set to be caught by the calling process are set to the default action in the child process.
- Signals (including SIGCHLD) set to be ignored by the calling process are set to be ignored by the child process.
- The child process uses the same scheduling policy and parameters as the parent.
- The child process inherits the parent's effective user and group IDs.

If the `setuid` bit is set for the process image file, the child process's effective user ID becomes the file's owner ID before the process starts. Similarly, if the `setgid` bit is set for the process image file, the child process's effective group ID becomes the file's group ID before the process starts.

- The child process is started on the same node that the calling process is running on.
- The child runs in the same adaptive partition as the parent.
- The child's runmask is set to its parent's inherit mask.
- There's no maximum size for the child process's stack.
- The child becomes a zombie on its death.
- The child isn't held for debugging purposes.
- The child uses the system default settings for alignment.
- The child uses Address Space Layout Randomization (ASLR) if the parent uses it.

If the value of the *attrp* pointer is `NULL`, then the default values are used.



All process attributes—other than those influenced by the spawn attributes or the file actions—appear in the new process image as though *fork()* had been called to create a child process, and then a member of the *exec()* family of functions had been called by the child process to execute the new process image.

In the QNX Neutrino implementation, fork handlers aren't run when you call *posix_spawn()* or *posix_spawnnp()*.

You can do the following by using the attributes defined by POSIX. You can get and set the POSIX flags by calling the POSIX [posix_spawnattr_getflags\(\)](#) (p. 2068) and [posix_spawnattr_setflags\(\)](#) (p. 2098) routines, or the QNX Neutrino [posix_spawnattr_getxflags\(\)](#) (p. 2090) and [posix_spawnattr_setxflags\(\)](#) (p. 2119) routines.

If you want to:	Do the following:
Explicitly specify the process group for the child process	Set the POSIX_SPAWN_SETPGROUP flag, and call posix_spawnattr_setpgroup() (p. 2103), specifying a nonzero process group ID.
Put the child in a new process group with a process group ID equal to its process ID	Set the POSIX_SPAWN_SETPGROUP flag, and call posix_spawnattr_setpgroup() (p. 2103), specifying a process group ID of zero.
Set the initial signal mask of the child process	Set POSIX_SPAWN_SETSIGMASK and call posix_spawnattr_setsigmask() (p. 2115), specifying the set of signals you want to mask
Set the actions for certain signals to the default action in the child process	Set POSIX_SPAWN_SETSIGDEF and call posix_spawnattr_setsigdefault() (p. 2111), specifying the set of signals.
Explicitly set the child process's scheduling policy and parameters	Set the POSIX_SPAWN_SETSCHEDULER flag (in which case the setting of the POSIX_SPAWN_SETSCHEDPARAM flag doesn't matter), call posix_spawnattr_setschedpolicy() (p. 2109) to specify the scheduling policy, and call posix_spawnattr_setschedparam() (p. 2107) to specify the scheduling parameters.
Set the child process's scheduling parameters, but use the parent's scheduling policy	Set the POSIX_SPAWN_SETSCHEDPARAM flag and call posix_spawnattr_setschedparam() (p. 2107) to specify the scheduling parameters.
Reset the child process's effective user and group IDs to the parent's real user and group IDs, and make the child process inherit the parent's real (rather than the effective) user and group IDs.	Set the POSIX_SPAWN_RESETPID flag. If the setuid bit is set for the process image file, the child process's effective user ID becomes the file's owner ID before

If you want to:	Do the following:
	the process starts. Similarly, if the setgid bit is set for the process image file, the child process's effective group ID becomes the file's group ID before the process starts.

You can do the following by using the attributes defined as extensions by QNX Neutrino. You can get and set the QNX Neutrino *and* POSIX flags by calling [posix_spawnattr_getxflags\(\)](#) (p. 2090) and [posix_spawnattr_setxflags\(\)](#) (p. 2119). Both of these routines are QNX Neutrino extensions.

If you want to:	Do the following:
Spawn the child process on a different node	Set POSIX_SPAWN_SETND and call posix_spawnattr_setnode() (p. 2101), specifying the remote node ID
Make the child process ignore specific signals	Set POSIX_SPAWN_SETSIGIGN and call posix_spawnattr_setsigignore() (p. 2113), specifying the signals you want to ignore
Associate the child process with a scheduler partition	Set POSIX_SPAWN_SETSPART and call posix_spawnattr_addpartid() (p. 2059), specifying a partition ID, or posix_spawnattr_addpartition() (p. 2061), specifying a partition name
Restrict the CPUs that the child process can run on	Set POSIX_SPAWN_EXPLICIT_CPU and call posix_spawnattr_setrunmask() (p. 2105), specifying the CPU affinity/runmask. If you don't set flag, the child inherits the inherit mask of the calling thread.
Specify the user and group IDs for the child process	Set POSIX_SPAWN_SETCRED and call posix_spawnattr_setcred() (p. 2095), specifying the IDs
Set the maximum stack size for the child process	Set POSIX_SPAWN_SETSTACKMAX and call posix_spawnattr_setstackmax() (p. 2117), specifying the stack size. If you don't set POSIX_SPAWN_SETSTACKMAX, there's no maximum. You can retrieve the default value with a call to

If you want to:	Do the following:
	<i>posix_spawnattr_getstackmax()</i> (p. 2088) using an initialized <code>posix_spawnattr_t</code> object.
Assign a new application ID to the process.	Set <code>POSIX_SPAWN_NEWAPP</code> . If you don't set this flag, the process inherits the parent's application ID.
Prevent the child process from becoming a zombie on its death	Set <code>POSIX_SPAWN_NOZOMBIE</code> . No child return or exit information will be available.
Hold the child process for debugging purposes	Set <code>POSIX_SPAWN_HOLD</code>
Use the system default settings for alignment	Set <code>POSIX_SPAWN_ALIGN_DEFAULT</code>
Attempt to always fault data misalignment references	Set <code>POSIX_SPAWN_ALIGN_FAULT</code>
Make the child process not fault on misalignment; attempt to fix it instead (this approach might be slow)	Set <code>POSIX_SPAWN_ALIGN_NOFAULT</code>
Toggle the Address Space Layout Randomization bit. If the parent has ASLR, turn it off for the child; if the parent's bit is off, turn it on for the child.	Set <code>POSIX_SPAWN_ASLR_INVERT</code>
Make the new process a session leader	Set <code>POSIX_SPAWN_SETSID</code>
Debug process (this is used only for debugging the kernel itself)	Set <code>POSIX_SPAWN_DEBUG</code>
Cause the spawn to act like <code>exec*()</code> : replace the calling program in memory with the newly loaded program. If successful, no return is made to the calling program.	Set <code>POSIX_SPAWN_EXEC</code>
Search the <i>PATH</i> environment variable for the executable	Set <code>POSIX_SPAWN_SEARCH_PATH</code>
Start a shell, passing <i>path</i> as a script	Set <code>POSIX_SPAWN_CHECK_SCRIPT</code>
Indicate that the process is known to operate safely with 64-bit addressing or	Set <code>POSIX_SPAWN_PADDR64_SAFE</code>

If you want to:	Do the following:
doesn't care about the physical memory location	
Start a new terminal group	Set POSIX_SPAWN_TCSETPGROUP

Returns:

EOK

The child was successfully spawned.

EINVAL

An argument was invalid; the spawn attributes or file actions object is invalid, or the name of the process image file is **NULL** or an empty string.

If you specify an invalid value in the file actions or spawn attributes, *posix_spawn()* or *posix_spawnp()* might fail; if the value causes an error to occur after the child has been successfully spawned, the child process may exit with a status of 127.

EIO

An internal error occurred in the library.

ENOENT

- For *posix_spawn()*, the executable specified by the *path* argument couldn't be found. The *path* must be the full path to the executable. Use *posix_spawnp()* if you want to search for the executable to spawn.
- For *posix_spawnp()*, the executable specified by the *file* argument couldn't be found in any of the **PATH** environment variable locations.

ENOMEM

The memory required to create the message to send to *procnto* couldn't be allocated memory to create the new process and its associated data structures couldn't be allocated. For partitions, the partition ID couldn't be added to the attributes object.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set **PROT_EXEC** for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

If *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause [fork\(\)](#) (p. 685) or one of the [exec*\(\)](#) (p. 487) family of functions to fail, an error value is returned as described for those functions (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If `POSIX_SPAWN_SETPGROUP` is set in the *spawn-flags* attribute of the object referenced by *attrp*, and *posix_spawn()* or *posix_spawnp()* fails while changing the child's process group, an error value is returned as described for [setpgid\(\)](#) (p. 2886) (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If `POSIX_SPAWN_SETSCHEDPARAM` is set and `POSIX_SPAWN_SETSCHEDULER` isn't set in the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause [sched_setparam\(\)](#) (p. 2738) to fail, an error value is returned as described for *sched_setparam()* (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If `POSIX_SPAWN_SETSCHEDULER` is set in the *spawn-flags* attribute of the object referenced by *attrp*, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause [sched_setscheduler\(\)](#) (p. 2740) to fail, an error value is returned as described for *sched_setscheduler()* (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If the *file_actions* argument isn't `NULL`, and specifies any [close\(\)](#) (p. 336), [dup2\(\)](#) (p. 485), or [open\(\)](#) (p. 1905) actions to be performed, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause *close()*, *dup2()*, or *open()* to fail, an error value is returned as described for those functions (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127). For example, if the parent process closes a file descriptor while it's being duplicated, you'll get an error of `EBADF`. An *open()* file action may, by itself, result in any of the errors described by *close()* or *dup2()*, in addition to those described by *open()*.

The *posix_spawn()* and *posix_spawnp()* functions can also return any error returned by a [stat\(\)](#) (p. 3164) on *path*.

When Adaptive Partitioning modules are also included in the image, the following error codes can be returned:

EACCES

The spawned program doesn't have permission to associate with the specified partitions.

EEXIST

You tried to associate the child process with more than one scheduling partition. This typically occurs when you create a group name partition with more than one scheduler pseudo-partition, and then attempt to spawn a process and associate with that group name partition.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_spawn_file_actions_addclose()

Add a “close a file descriptor” action to a spawn file actions object

Synopsis:

```
#include <spawn.h>

int posix_spawn_file_actions_addclose(
    posix_spawn_file_actions_t *fact_p,
    int fd);
```

Arguments:

fact_p

A pointer to the spawn file actions object that you want to update. You must have already initialized this object by calling [*posix_spawn_file_actions_init\(\)*](#) (p. 2057).

fd

The file descriptor that you want to close when the child process starts.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawn_file_actions_addclose()* function adds a close action to a spawn file actions object (the object referenced by *fact_p*) that causes the file descriptor *fd* to be closed (as if `close(fd)` had been called) when a new process is spawned using this file actions object.

Returns:

EOK

Success.

EINVAL

An argument was invalid, the value specified by *fact_p* was invalid, or the value specified by *fd* was negative or greater than or equal to `{OPEN_MAX}`.

ENOMEM

The action couldn't be added to the file actions object, or insufficient memory exists to add to the spawn file actions object.

It isn't considered an error if the *fd* argument specifies a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn()* or *posix_spawnnp()* operation.

Classification:

[POSIX 1003.1 SPN](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawn_file_actions_adddup2()

Add a “duplicate a file descriptor” action to a spawn file actions object

Synopsis:

```
#include <spawn.h>

int posix_spawn_file_actions_adddup2(
    posix_spawn_file_actions_t *fact_p,
    int fd,
    int new_fd);
```

Arguments:

fact_p

A pointer to the spawn file actions object that you want to update. You must have already initialized this object by calling [*posix_spawn_file_actions_init\(\)*](#) (p. 2057).

fd

The file descriptor that you want to duplicate.

new_fd

The number that you want to use for the new file descriptor.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawn_file_actions_adddup2()* function adds a *dup2()* action to the object referenced by *fact_p* that causes the file descriptor *fd* to be duplicated as *new_fd* (as if a call were made to `dup2(fd, new_fd)`) when a new process is spawned using this file actions object.

Returns:

EOK

Success.

EBADF

The value specified by *fd* or *new_fd* is negative or greater than or equal to *{OPEN_MAX}*.

EINVAL

An argument was invalid. The value specified by file actions is invalid.

ENOMEM

The action couldn't be added to the file actions object, or insufficient memory exists to add to the spawn file actions object.

It isn't considered an error for the *fd* argument to specify a file descriptor for which the specified operation couldn't be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn()* or *posix_spawnnp()* operation.

Classification:

[POSIX 1003.1 SPN](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawn_file_actions_addopen()

Add an “open a file” action to a spawn file action object

Synopsis:

```
#include <spawn.h>

int posix_spawn_file_actions_addopen(
    posix_spawn_file_actions_t *_Restrict fact_p,
    int new_fd,
    const char *_Restrict path,
    int oflags,
    mode_t omode);
```

Arguments:

fact_p

A pointer to the spawn file actions object that you want to update. You must have already initialized this object by calling [posix_spawn_file_actions_init\(\)](#) (p. 2057).

new_fd

The number that you want to use for the new file descriptor.

path

The name of the file to open. The string described by *path* is copied by the *posix_spawn_file_actions_addopen()* function.

oflags

Flags that specify the status and access modes of the file.

omode

An object of type `mode_t` that specifies the access mode to use for a newly created file. For more information, see “Access permissions” and “O_CREATE” in the documentation for [stat\(\)](#) (p. 3164).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawn_file_actions_addopen()` function adds an open action to the object referenced by `fact_p` that causes the file named by `path` to be opened (as if `open(path, oflag, mode)` had been called, and the returned file descriptor, if not already `new_fd`, had been changed to `new_fd`) when a new process is spawned using this file actions object. If `new_fd` was already an open file descriptor, it will be closed before the new file is opened.

Files that need to be opened for use by the spawned process can alternatively be handled either by having the calling process open them before the `posix_spawn()` or `posix_spawnnp()` call (and close them after), or by passing filenames to the spawned process (in `argv`) so that it may open them itself.

Note that the spawn open file action provides to `posix_spawn()` and `posix_spawnnp()` the same capability that the shell redirection operators provide to `system()`, only without the intervening execution of a shell; for example:

```
system ("myprog <file1 3<file2");
```

If the calling process needs to open one or more files for access by the spawned process, but has insufficient spare file descriptors, then the open action is necessary to allow the `open()` to occur in the context of the child process after other file descriptors have been closed (that must remain open in the parent). Additionally, if a parent is executed from a file having a “set-user-id” mode bit set and the `POSIX_SPAWN_RESETIDS` flag is set in the spawn attributes (see [posix_spawn\(\)](#) (p. 2037)), a file created within the parent process will (possibly incorrectly) have the parent's effective user ID as its owner, whereas a file created with an `open()` action during `posix_spawn()` or `posix_spawnnp()` will have the parent's real ID as its owner; an open action by the parent process may successfully open a file to which the real user should not have access, or fail to open a file to which the real user should have access.

Returns:**EOK**

Success.

EBADF

The value specified by `new_fd` is negative or greater than or equal to `{OPEN_MAX}`.

EINVAL

An argument was invalid, or the value specified by `fact_p` was invalid.

ENOMEM

The partition ID couldn't be added to the attributes object, or insufficient memory exists to add to the spawn file actions object.

It isn't considered an error if the *new_fd* argument specifies a file descriptor for which the specified operation couldn't be performed at the time of the call. This type of error isn't detected when the associated file actions object is later used during a *posix_spawn()* or *posix_spawnp()* operation.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawn_file_actions_destroy()

Destroy a spawn file actions object

Synopsis:

```
#include <spawn.h>

int posix_spawn_file_actions_destroy(
    posix_spawn_file_actions_t *fact_p);
```

Arguments:

fact_p

A pointer to the spawn file action object that you want to destroy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawn_file_actions_destroy()` function destroys the object referenced by `fact_p`; in effect, the object becomes uninitialized.



The `posix_spawn_file_actions_destroy()` function renders the `posix_spawn_file_actions_t` unusable. Before reusing the object, you must initialize it with `posix_spawn_file_actions_init()`. POSIX specifies that results are undefined when you use a `posix_spawnattr_t` object after it has been destroyed, including calling `posix_spawn_file_actions_destroy()` on an already destroyed object.

Returns:

EOK

Success.

EINVAL

The value specified by `fact_p` was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawn_file_actions_init()

Initialize a spawn file actions object

Synopsis:

```
#include <spawn.h>

int posix_spawn_file_actions_init(
    posix_spawn_file_actions_t *fact_p);
```

Arguments:

fact_p

A pointer to the spawn file action object that you want to initialize.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawn_file_actions_init()` function initializes the object referenced by `fact_p` to contain no file actions for `posix_spawn()` or `posix_spawnnp()` to perform. For information about destroying a spawn file actions object, see [posix_spawn_file_actions_destroy\(\)](#) (p. 2055).

A spawn file actions object is an opaque data type that specifies a series of `close()`, `dup2()`, and `open()` actions to be performed by a `posix_spawn()` or `posix_spawnnp()` operation in order to arrive at the set of open file descriptors for the child process, given the parent's set of open file descriptors. For more information, see [posix_spawn\(\)](#) (p. 2037)).



Don't reinitialize a `posix_spawn_file_actions_t` object without destroying it. Repeated calls to `posix_spawn_file_actions_init()` with the same `posix_spawn_file_actions_t` object could result in a memory leak. In the QNX Neutrino implementation, all get and set operations, as well as `posix_spawn()`, return `EINVAL` when attempting to reference a destroyed file actions object.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

ENOMEM

Insufficient memory exists to initialize the spawn file actions object.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_addpartid()

Specify the ID of a resource partition that the spawned process should be associated with

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_addpartid(
    posix_spawnattr_t *_Restrict attrp,
    part_id_t part_id,
    part_dcmd_flags_t part_flags);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

part_id

A unique partition identifier for the partition that you want to associate the spawned process with. You can retrieve the ID by sending the appropriate *devctl()* command (see below) to the partition manager.

part_flags

The flags you want to associate with the partition. No flags are currently defined, so pass 0 for this argument.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_addpartid()* specifies, by its partition identifier, a resource partition that the spawned process should be associated with. You must have already initialized the spawn attributes object by calling *posix_spawnattr_init()* (p. 2093). If you specify an ID for a non-existent partition, it won't be known until the *posix_spawn()* call is made.

An additional form for this call, *posix_spawnattr_addschedpartid()*, is implemented as a macro in *<spawn.h>*.

The partition association takes effect only if the `POSIX_SPAWN_SETSPART` flag is set in the spawn attributes; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). Setting this flag without specifying a scheduling partition may produce unintended results.



This function is a QNX Neutrino extension.

For information about inheritance rules as they apply to partition association, see the *Adaptive Partitioning User's Guide*.

Returns:

- EOK**
- Success.
- EINVAL**
- An argument was invalid.
- ENOMEM**
- The partition ID couldn't be added to the attributes object.



Some partition association errors may not be reported until the `posix_spawn()` call is made.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_addpartition()

Specify the name of a resource partition that the spawned process should be associated with

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_addpartition(
    posix_spawnattr_t *_Restrict attrp,
    const char *partition_path,
    part_dcnd_flags_t part_flags);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

partition_path

A path to the partition. Given a full partition path, determine the path type (memory/sched/ etc.) and return a pointer to the name suffix (i.e. the portion of the name after /partition/mem/ or /partition/sched/), if possible.

part_flags

The flags you want to associate with the partition. No flags are currently defined, so pass 0 for this argument.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_addpartition()* function specifies a resource partition that the spawned process should be associated, and it's the most convenient form of adding partition associations because it accepts a partition name rather than an ID. You must have already initialized the spawn attributes object by calling [*posix_spawnattr_init\(\)*](#) (p. 2093).

Partition names can either be group names, pseudo-partitions or real partitions, and are always resolved to a “real” partition; otherwise, an error is returned.



If *partition_path* doesn't refer to a real partition, then the *part_flags* argument applies to all resolved real partitions referred to by the group name or pseudo-partition. This may result in an error if the flag doesn't apply to the partition type (as reported by the *PART_TYPE()* macro defined in `<sys/part.h>`).

The partition association takes effect only if the `POSIX_SPAWN_SETSPART` flag is set in the spawn attributes; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). Setting this flag without specifying a scheduling partition may produce unintended results.

For information about inheritance rules as they apply to partition association, see the *Adaptive Partitioning Users Guide*.



This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

ENOMEM

The partition ID couldn't be added to the attributes object.



If this function fails and the *partition_name* provided is a group name, then it's unspecified how many of the pseudo-partitions within the group name are added to the attributes object pointed to by *attrp*. For this scenario, we recommend that you destroy the attributes object, and initialize a new attributes object.



Some partition association errors may not be reported until the *posix_spawn()* call is made.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_destroy()

Destroy a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attrp);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to destroy.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_destroy()* function destroys a spawn attributes object. You can reinitialize a destroyed *attrp* attributes object by using [posix_spawnattr_init\(\)](#) (p. 2093); the results of referencing the object after you've destroyed it are undefined.

A spawn attributes object is an opaque object of type `posix_spawnattr_t` (defined in `<spawn.h>`), and it specifies the inheritance of process attributes across a spawn operation.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of *posix_spawn()* or *posix_spawnnp()*. Changes that you make to the spawn attributes object don't affect any processes that you've already spawned.



The *posix_spawnattr_destroy()* function renders the `posix_spawnattr_t` unusable. Before you can reuse the object, you must initialize it with *posix_spawnattr_init()*. POSIX specifies that results are undefined if you use a `posix_spawnattr_t` object after it has been destroyed, including calling *posix_spawnattr_destroy()* on an already destroyed object. If you attempt to reference a destroyed attributes object, all of the *posix_spawnattr_get*()* and *posix_spawnattr_set*()* operations, as well as *posix_spawn()*, return `EINVAL`.

Returns:

EOK

Success.

EINVAL

The value specified by *attrp* is invalid.

Classification:

[POSIX 1003.1 SPN](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getcred()

Get the user and group ID attributes from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getcred(
    const posix_spawnattr_t *_Restrict attrp,
    uid_t *_Restrict uid_p,
    gid_t *_Restrict gid_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

uid_p

A pointer to a location where the routine can store the current user ID attribute.

gid_p

A pointer to a location where the routine can store the current group ID attribute.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getcred()* function retrieves the current values of the credential attributes (user and group IDs) from the spawn attribute object pointed to by *attrp*, and stores them in the locations pointed to by *uid_p* and *gid_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

These attributes are used as the user and group IDs of the child process created by a spawn operation if `POSIX_SPAWN_SETCRED` is set in the spawn flags; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119).

To set the value of these attributes, call [posix_spawnattr_setcred\(\)](#) (p. 2095).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getflags()

Get the POSIX flags from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getflags(
    const posix_spawnattr_t *_Restrict attrp,
    short *_Restrict flags_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

flags_p

A pointer to a location where the function can store the flags.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getflags()* function retrieves the POSIX flags from the spawn attributes object pointed to by *attrp*, and stores them in the object pointed to by *flags_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

These flags indicate which process attributes you want to change in the new process image when you invoke *posix_spawn()* or *posix_spawnnp()*. For detailed information about the flags, see the entry for [posix_spawn\(\)](#) (p. 2037).

This function doesn't get the extended flags that QNX Neutrino defines:



- POSIX_SPAWN_ALIGN_DEFAULT
- POSIX_SPAWN_ALIGN_FAULT
- POSIX_SPAWN_ALIGN_NOFAULT
- POSIX_SPAWN_ASLR_INVERT

- POSIX_SPAWN_CHECK_SCRIPT
- POSIX_SPAWN_DEBUG
- POSIX_SPAWN_EXEC
- POSIX_SPAWN_EXPLICIT_CPU
- POSIX_SPAWN_HOLD
- POSIX_SPAWN_NEWAPP
- POSIX_SPAWN_NOZOMBIE
- POSIX_SPAWN_PADDR64_SAFE
- POSIX_SPAWN_SEARCH_PATH
- POSIX_SPAWN_SETCRED
- POSIX_SPAWN_SETND
- POSIX_SPAWN_SETSID
- POSIX_SPAWN_SETSIGIGN
- POSIX_SPAWN_SETSPART
- POSIX_SPAWN_SETSTACKMAX
- POSIX_SPAWN_TCSETPGROUP

To get the extended flags, use [posix_spawnattr_getxflags\(\)](#) (p. 2090).

Returns:

EOK

Success.

EINVAL

The value specified by *attrp* or *flags_p* is invalid.

Classification:

[POSIX 1003.1 SPN](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getnode()

Get the remote node attribute from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getnode(
    const posix_spawnattr_t *_Restrict attrp,
    uint32_t *_Restrict node_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

node_p

A pointer to a location where the routine can store the node descriptor of the remote node on which to spawn the child process.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getnode()* function retrieves the remote node attribute from the spawn attributes object referenced by *attrp*, and stores it in the object pointed to by *node_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute specifies the descriptor of the node on which the child process is to be spawned if `POSIX_SPAWN_SETND` is set in the spawn flags; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). By default, the child is spawned on the node on which you call *posix_spawn()*.

To set the value of this attribute, call [posix_spawnattr_setnode\(\)](#) (p. 2101).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:**EOK**

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getpartid()

Retrieve the partition identifiers currently set in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getpartid(
    const posix_spawnattr_t *_Restrict attrp,
    int *num,
    partlist_info_t plist_info[]);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

num

A pointer to a location that specifies the number of elements in the *plist_info* array, and where the function can store information about the number of elements required (see below).

plist_info

An array of `partlist_info_t` objects (see `<sys/part.h>`) where the function can store information about the partitions.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getpartid()* function retrieves the partition IDs that are currently stored in a spawn attributes object. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

The partition association takes effect only if the `POSIX_SPAWN_SETSPART` flag is set in the spawn attributes; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). Setting this flag without specifying a scheduling partition may produce unintended results.

You must provide the storage for the `partlist_info_t` array, and you must set the value pointed to by `num` to be the number of elements in the list.

The function sets the value pointed to by `num` as follows:

- If the actual number of entries is less than or equal to the number for which you allocated space, then the function fills in the number of entries and sets the number that `num` points to the actual number of entries.
- If the actual number of entries is greater than the number for which you allocated space, then the function fills in `*num` entries and sets the number pointed to by `num` to the negative of the number of entries that wouldn't fit.

For example, if you provide space for four entries and set `*num` to 4, but there are six entries in the `posix_spawnattr_t` object, then the function places four entries into the structure and sets the number pointed to by `num` to -2.

To determine how many entries you need, set `*num` to 0 and call `posix_spawnattr_getpartid()`. You can then allocate enough space to retrieve all of them.



- You can distinguish partition ID types with the `PART_TYPE()` macro defined in `<sys/part.h>`.
- This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getpgroup()

Get the process group attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getpgroup(
    const posix_spawnattr_t *_Restrict attrp,
    pid_t *_Restrict pid_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

pid_p

A pointer to a location where the function can store the group ID.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getpgroup()* function gets the value of the process group attribute from the attributes object pointed to by *attrp*, and stores it in the object pointed to by *pid_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute represents the process group to be joined by the new process image in a spawn operation if `POSIX_SPAWN_SETPGROUP` is set in the spawn flags; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default value of the process group attribute is zero.

You can use [posix_spawnattr_setpgroup\(\)](#) (p. 2103). to set it.

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success

EINVAL

The value specified by *attrp* or *pid_p* was invalid.

Classification:

[POSIX 1003.1 SPN](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getrunmask()

Get the runmask attribute from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getrunmask(
    const posix_spawnattr_t *_Restrict attrp,
    uint32_t *_Restrict runmask_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

runmask_p

A pointer to a location where the routine can store the current runmask.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getrunmask()* routine retrieves the current value of the runmask attribute from the given spawn attributes object. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute is a bitmask of the processors that the spawned program can run on. Processor numbering starts at 0, with each processor corresponding to a bit in the runmask. For example, a runmask of 0x00000005 indicates that the spawned process is restricted to CPUs 0 and 2. For more information, see the Multicore Processing *User's Guide*.

This setting takes effect only if the `POSIX_SPAWN_EXPLICIT_CPU` flag is set; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default mask is 0xFFFFFFFFFU, indicating that the spawned process can run on any processor.

To set the value of this attribute, call [posix_spawnattr_setrunmask\(\)](#) (p. 2105).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getschedparam()

Get the scheduling parameters attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getschedparam(
    const posix_spawnattr_t *_Restrict attrp,
    struct sched_param *_Restrict sched_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

sched_p

A pointer to a [sched_param](#) (p. 2731) structure where the function can store the scheduling parameters.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getschedparam()* function obtains the value of the scheduling parameters attribute from the attributes object pointed to by *attrp*, and stores the value in the location pointed to by *sched_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

The scheduling parameters attribute governs the parameter assigned to a new process image in a spawn operation if `POSIX_SPAWN_SETSCHEDULER` or `POSIX_SPAWN_SETSCHEDPARAM` is set in the spawn flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default value of the process group attribute is zero. By default, the child process uses the same scheduling parameters as the parent.

You can specify the value of this attribute by calling [posix_spawnattr_setschedparam\(\)](#) (p. 2107).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getschedpolicy()

Get the scheduling policy attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getschedpolicy(
    const posix_spawnattr_t *_Restrict attrp,
    int *_Restrict policy_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

policy_p

A pointer to a location where the routine can store the scheduling policy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getschedpolicy()* function obtains the value of the scheduling policy attribute from the attributes object referenced by *attrp* and stores it in the location pointed to by *policy_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute specifies the scheduling policy to assign to the new process image in a spawn operation if `POSIX_SPAWN_SETSCHEDULER` is set in the spawn flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default value of the process group attribute is zero. By default, the child process uses the same scheduling policy as the parent.

You can specify the value of this attribute by calling [posix_spawnattr_setschedpolicy\(\)](#) (p. 2109).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getsigdefault()

Get the “default signals” attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getsigdefault(
    const posix_spawnattr_t *_Restrict attrp,
    sigset_t *_Restrict sigset_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

sigset_p

A pointer to a location where the routine can store the set of signals that are to be handled as per the default behavior.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getsigdefault()* function obtains the value of the “default signals” attribute from the attributes object referenced by *attrp*, and stores it in the signal set pointed to by *sigset_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute represents the set of signals that will use the default signal handling in the new process image created by a spawn operation if `POSIX_SPAWN_SETSIGDEF` is set in the spawn-flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119).

To set the value of this attribute, call [posix_spawnattr_setsigdefault\(\)](#) (p. 2111).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getsigignore()

Get the “ignored signals” attribute from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getsigignore(
    const posix_spawnattr_t *_Restrict attrp,
    sigset_t *_Restrict sigset_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

sigset_p

A pointer to a location where the routine can store the current set of signals to be ignored.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getsigignore()* routine retrieves the “ignored signals” attribute from the given spawn attributes object. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute specifies the set of signals that should be ignored by the spawned process. This setting takes only effect if the `POSIX_SPAWN_SETSIGIGN` flag is set in the spawn attributes; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119).

To set the value of this attribute, call [posix_spawnattr_setsigignore\(\)](#) (p. 2113).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getsigmask()

Get the signal mask attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getsigmask(
    const posix_spawnattr_t *_Restrict attrp,
    sigset_t *_Restrict sigset_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

sigset_p

A pointer to a location where the routine can store the current value of the signal mask attribute.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getsigmask()* function retrieves the signal mask attribute from the spawn attributes object referenced by *attrp*, and stores it in the object pointed to by *sigset_p*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

The value of this attribute is used as the signal mask in the new process image of a spawn operation if `POSIX_SPAWN_SETSIGMASK` is set in the spawn flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119).

To set the value of this attribute, call [posix_spawnattr_setsigmask\(\)](#) (p. 2115).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getstackmax()

Get the maximum stack size attribute from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getstackmax(
    const posix_spawnattr_t *_Restrict attrp,
    uint32_t *_Restrict size_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

size_p

A pointer to a location where the routine can store the current maximum stack size.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_getstackmax()` function retrieves the maximum stack size attribute from the given spawn attributes object. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This setting takes effect only if the `POSIX_SPAWN_SETSTACKMAX` flag is set in the spawn attributes; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default is no maximum.

To set the value of this attribute, call [posix_spawnattr_setstackmax\(\)](#) (p. 2117).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_getxflags()

Get the POSIX flags and the QNX Neutrino extended flags from a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_getxflags(
    const posix_spawnattr_t *_Restrict attrp,
    uint32_t *_Restrict flags_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to query.

flags_p

A pointer to a location where the function can store the flags.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_getxflags()* function retrieves the POSIX flags and QNX Neutrino extended flags from the spawn attributes object pointed to by *attrp*, and stores them in the object pointed to by *flags_p*. You must have already initialized the spawn attributes object by calling [*posix_spawnattr_init\(\)*](#) (p. 2093).

POSIX defines the following flags:

- POSIX_SPAWN_SETPGROUP
- POSIX_SPAWN_SETSIGMASK
- POSIX_SPAWN_SETSIGDEF
- POSIX_SPAWN_SETSCHEDULER
- POSIX_SPAWN_SETSCHEDPARAM
- POSIX_SPAWN_RESETIDS

QNX Neutrino defines the following extended flags:

- POSIX_SPAWN_ALIGN_DEFAULT
- POSIX_SPAWN_ALIGN_FAULT
- POSIX_SPAWN_ALIGN_NOFAULT
- POSIX_SPAWN_ASLR_INVERT
- POSIX_SPAWN_CHECK_SCRIPT
- POSIX_SPAWN_DEBUG
- POSIX_SPAWN_EXEC
- POSIX_SPAWN_EXPLICIT_CPU
- POSIX_SPAWN_HOLD
- POSIX_SPAWN_NEWAPP
- POSIX_SPAWN_NOZOMBIE
- POSIX_SPAWN_PADDR64_SAFE
- POSIX_SPAWN_SEARCH_PATH
- POSIX_SPAWN_SETCRED
- POSIX_SPAWN_SETND
- POSIX_SPAWN_SETSID
- POSIX_SPAWN_SETSIGIGN
- POSIX_SPAWN_SETSPART
- POSIX_SPAWN_SETSTACKMAX
- POSIX_SPAWN_TCSETPGROUP

These flags indicate which process attributes you want to change in the new process image when you invoke *posix_spawn()* or *posix_spawnnp()*. For detailed information about the flags, see the entry for [posix_spawn\(\)](#) (p. 2037).



The *posix_spawnattr_getxflags()* function is a QNX Neutrino extension. POSIX specifies the [posix_spawnattr_getflags\(\)](#) (p. 2068) and [posix_spawnattr_setflags\(\)](#) (p. 2098) routines, which get and set only the POSIX flags.

Returns:

EOK

Success.

EINVAL

The value specified by *attrp* or *flags_p* is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_init()

Initialize a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_init( posix_spawnattr_t *attrp );
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to initialize.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_init()` function initializes the given spawn attributes object with the default value for all the attributes. A spawn attributes object is an opaque object of type `posix_spawnattr_t` (defined in `<spawn.h>`), and it specifies the inheritance of process attributes across a spawn operation.

You can also use this function to reinitialize a spawn attributes object that you've destroyed by calling [`posix\_spawnattr\_destroy\(\)`](#) (p. 2064).

-
- The `posix_spawnattr_destroy()` function renders the `posix_spawnattr_t` object unusable. Before you can reuse it (which includes calling `posix_spawnattr_destroy()` on an already destroyed object), you must reinitialize the object with `posix_spawnattr_init()`.



- POSIX indicates that the results of reinitializing an already initialized `posix_spawnattr_t` object are undefined. The QNX Neutrino implementation doesn't support the reinitialization of a `posix_spawnattr_t` object without an intervening destruction of the object. Repeated calls to `posix_spawnattr_init()` with the same `posix_spawnattr_t` object could result in a memory leak.
-

Returns:**EOK**

Success.

ENOMEM

Insufficient memory exists to initialize the spawn attributes object.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setcred()

Set the user and group ID attributes in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setcred(
    posix_spawnattr_t *attrp,
    uid_t uid,
    gid_t gid);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

uid

The user ID that you want to use for spawned processes.

gid

The group ID that you want to use for spawned processes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_setcred()* function sets the values of the credential attributes (user and group IDs) in the spawn attribute object pointed to by *attrp*. You must have already initialized the spawn attributes object by calling [*posix_spawnattr_init\(\)*](#) (p. 2093).

These attributes are used as the user and group IDs of the child process created by a spawn operation if `POSIX_SPAWN_SETCRED` is set in the spawn flags; to set this flag, call [*posix_spawnattr_setxflags\(\)*](#) (p. 2119). By default, a process spawned with *posix_spawn()* or *posix_spawnnp()* inherits the credentials from the parent process.

You must provide both a user ID (*uid*) and group ID (*gid*), but you can effectively leave one of them unchanged by passing the results of [*getuid\(\)*](#) (p. 984) or [*getgid\(\)*](#) (p. 849), respectively.

In order for the *uid/gid* to take effect:

- You must pass the `posix_spawnattr_t` attributes object as a parameter to `posix_spawn()`.
- You must have set `POSIX_SPAWN_SETCRED` in the spawn flags by calling [posix_spawnattr_setxflags\(\)](#) (p. 2119).
- The calling process must have the `PROCGR_AID_SPAWN_SETUID` or `PROCGR_AID_SPAWN_SETGID` ability (or both) set; see [procgr_ability\(\)](#) (p. 2141).

If these criteria are met, the real and effective user and group IDs of the spawned process are set to the specified credentials; however, the *setuid/setgid* mode of the spawned image takes precedence over the effective *uid/gid* of the spawned process in the same way as described for `POSIX_SPAWN_RESETIDS`. That is, the *setuid/setgid* mode causes the spawned process to inherit the *uid/gid* of the image as its effective *uid/gid*, respectively. The real *uid* and *gid* are as specified in the credentials.



The credentials of the spawned process take effect before the process actually runs, and therefore determine its access to specific resources. This may result in a failure in the spawned process, asynchronous to a successful `posix_spawn()` or `posix_spawnnp()` call.

If you set both `POSIX_SPAWN_SETCRED` and `POSIX_SPAWN_RESETIDS`, `POSIX_SPAWN_SETCRED` takes precedence.

To get the value of these attributes, call [posix_spawnattr_getcred\(\)](#) (p. 2066).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

posix_spawnattr_setflags()

Set the POSIX flags in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setflags(
    posix_spawnattr_t *attrp,
    short flags);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

flags

A bitwise OR of the POSIX flags you want to store in the spawn attributes:

- POSIX_SPAWN_SETPGROUP
- POSIX_SPAWN_SETSIGMASK
- POSIX_SPAWN_SETSIGDEF
- POSIX_SPAWN_SETSCHEDULER
- POSIX_SPAWN_SETSCHEDPARAM
- POSIX_SPAWN_RESETIDS

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_setflags()* function stores the POSIX flags in the spawn attributes object pointed to by *attrp*, overwriting any previously saved POSIX flags. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

These flags indicate which process attributes you want to change in the new process image when you invoke *posix_spawn()* or *posix_spawnnp()*. For detailed information about the flags, see the entry for [posix_spawn\(\)](#) (p. 2037).

You *can't* use this function to set the extended flags that QNX Neutrino defines:

- POSIX_SPAWN_ALIGN_DEFAULT
- POSIX_SPAWN_ALIGN_FAULT
- POSIX_SPAWN_ALIGN_NOFAULT
- POSIX_SPAWN_ASLR_INVERT
- POSIX_SPAWN_CHECK_SCRIPT
- POSIX_SPAWN_DEBUG
- POSIX_SPAWN_EXEC
- POSIX_SPAWN_EXPLICIT_CPU
- POSIX_SPAWN_HOLD
- POSIX_SPAWN_NEWAPP
- POSIX_SPAWN_NOZOMBIE
- POSIX_SPAWN_PADDR64_SAFE
- POSIX_SPAWN_SEARCH_PATH
- POSIX_SPAWN_SETCRED
- POSIX_SPAWN_SETND
- POSIX_SPAWN_SETSID
- POSIX_SPAWN_SETSIGIGN
- POSIX_SPAWN_SETSPART
- POSIX_SPAWN_SETSTACKMAX
- POSIX_SPAWN_TCSETPGROUP



To set the extended flags, use [posix_spawnattr_setxflags\(\)](#) (p. 2119). The *posix_spawnattr_setflags()* function doesn't affect any extended flags that you previously set in the spawn attributes object.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

[POSIX 1003.1 SPN](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setnode()

Set the remote node attribute in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setnode( posix_spawnattr_t *attrp,
                             uint32_t node);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

node

The identifier of the node that you want to spawn the process on.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_setnode()` function sets the remote node attribute in the spawn attributes object referenced by `attrp`. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute specifies the descriptor of the node on which the child process is to be spawned if `POSIX_SPAWN_SETND` is set in the spawn flags; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). By default, the child is spawned on the node on which you call `posix_spawn()`.

Use the [netmgr_strtnd\(\)](#) (p. 1883) function to obtain a valid node identifier for a named remote node.

To retrieve the value of this attribute, call [posix_spawnattr_getnode\(\)](#) (p. 2070).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:**EOK**

Success.

EINVAL

An argument was invalid.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setpgroup()

Set the process group attribute in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setpgroup(
    posix_spawnattr_t *attrp,
    pid_t pid);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

pid

The ID of the process group that you want the spawned process to join.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_setpgroup()* function sets the process group attribute in the spawn attributes object pointed to by *attr*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute represents the process group to be joined by the new process image in a spawn operation if `POSIX_SPAWN_SETPGROUP` is set in the spawn flags; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default value of the process group attribute is zero. The default value of the process group attribute is zero. You can use [posix_spawnattr_getpgroup\(\)](#) (p. 2074). to retrieve it.

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

ENOMEM

There wasn't enough memory available to extend the spawn attributes object.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setrunmask()

Set the runmask attribute in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setrunmask(
    posix_spawnattr_t *attrp,
    uint32_t runmask);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

runmask

The runmask that you want to use for the spawned processes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_getrunmask()` routine sets the value of the runmask attribute in the given spawn attributes object. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute is a bitmask of the processors that the spawned program can run on. Processor numbering starts at 0, with each processor corresponding to a bit in the runmask. For example, a runmask of 0x00000005 indicates that the spawned process is restricted to CPUs 0 and 2. For more information, see the *Multicore Processing User's Guide*.

This setting takes effect only if the `POSIX_SPAWN_EXPLICIT_CPU` flag is set; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default mask is 0xFFFFFFFFFU, indicating that the spawned process can run on any processor.

To get the value of this attribute, call [posix_spawnattr_getrunmask\(\)](#) (p. 2076).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:**EOK**

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setschedparam()

Set the scheduling parameters attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setschedparam(
    posix_spawnattr_t *_Restrict attrp,
    const struct sched_param *_Restrict sched_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

sched_p

A pointer to a [sched_param](#) (p. 2731) structure that holds the scheduling parameters that you want to use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_setschedparam()* function sets the value of the scheduling parameters attribute in the attributes object pointed to by *attrp*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

The scheduling parameters attribute governs the parameter assigned to a new process image in a spawn operation if `POSIX_SPAWN_SETSCHEDULER` or `POSIX_SPAWN_SETSCHEDPARAM` is set in the spawn flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default value of the process group attribute is zero. By default, the child process uses the same scheduling parameters as the parent.

You can retrieve the value of this attribute by calling [posix_spawnattr_getschedparam\(\)](#) (p. 2078).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:**EOK**

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setschedpolicy()

Set the scheduling policy attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setschedpolicy(
    posix_spawnattr_t *attrp,
    int policy);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

policy

The scheduling policy that you want to use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *posix_spawnattr_setschedpolicy()* function sets the scheduling policy attribute in the spawn attributes object pointed to by *attrp*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute specifies the scheduling policy to assign to the new process image in a spawn operation if `POSIX_SPAWN_SETSCHEDULER` is set in the spawn flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). The default value of the process group attribute is zero, meaning that the child process uses the same scheduling policy as the parent.

You can get the current value of this attribute by calling [posix_spawnattr_getschedpolicy\(\)](#) (p. 2080).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setsigdefault()

Set the “default signals” attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setsigdefault(
    posix_spawnattr_t *_Restrict attrp,
    const sigset_t *_Restrict sigset_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

sigset_p

A pointer to the set of signals that you want to be handled according to the default behavior.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_setsigdefault()` function sets the “default signals” attribute in the attributes object referenced by *attrp*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This attribute represents the set of signals that will use the default signal handling in the new process image created by a spawn operation if `POSIX_SPAWN_SETSIGDEF` is set in the spawn-flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119).

To retrieve the current value of this attribute, call [posix_spawnattr_getsigdefault\(\)](#) (p. 2082).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setsigignore()

Set the “ignored signals” attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setsigignore(
    posix_spawnattr_t *_Restrict attrp,
    const sigset_t *_Restrict sigset_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

sigset_p

A pointer to the set of signals that you want to be ignored.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_getsigignore()` routine sets the “ignored signals” attribute in the given spawn attributes object. You must have already initialized the spawn attributes object by calling `posix_spawnattr_init()` (p. 2093).

This attribute specifies the set of signals that should be ignored by the spawned process. You must have already initialized the spawn attributes object by calling `posix_spawnattr_init()` (p. 2093). This setting takes only effect if the `POSIX_SPAWN_SETSIGIGN` flag is set in the spawn attributes; to set this flag, call `posix_spawnattr_setxflags()` (p. 2119).

To get the value of this attribute, call `posix_spawnattr_getsigignore()` (p. 2084).

For more information about spawn attributes, see the entry for `posix_spawn()` (p. 2037).



This function is a QNX Neutrino extension.

Returns:**EOK**

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setsigmask()

Set the signal mask attribute of a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setsigmask(
    posix_spawnattr_t *_Restrict attrp,
    const sigset_t *_Restrict sigset_p);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

sigset_p

A pointer to the signal mask that you want to store.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_setsigmask()` function sets the signal mask attribute in an initialized attributes object referenced by *attrp*. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

The value of this attribute is used as the signal mask in the new process image of a spawn operation if `POSIX_SPAWN_SETSIGMASK` is set in the spawn flags attribute; to set this flag, call the POSIX function [posix_spawnattr_setflags\(\)](#) (p. 2098), or the QNX Neutrino function [posix_spawnattr_setxflags\(\)](#) (p. 2119). By default, the mask is an empty signal set.

To retrieve the current value of this attribute, call [posix_spawnattr_getsigmask\(\)](#) (p. 2086).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

POSIX 1003.1 SPN (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setstackmax()

Set the maximum stack size attribute in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setstackmax(
    posix_spawnattr_t *attrp,
    uint32_t size);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

size

The maximum stack size that you want to use for the spawned processes

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_setstackmax()` function sets the maximum stack size attribute in the given spawn attributes object. The default is no maximum. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

This setting takes effect only if the `POSIX_SPAWN_SETSTACKMAX` flag is set in the spawn attributes; to set this flag, call [posix_spawnattr_setxflags\(\)](#) (p. 2119).

To retrieve the value of this attribute, call [posix_spawnattr_getstackmax\(\)](#) (p. 2088).

For more information about spawn attributes, see the entry for [posix_spawn\(\)](#) (p. 2037).



This function is a QNX Neutrino extension.

Returns:

EOK

Success.

EINVAL

An argument was invalid.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

posix_spawnattr_setxflags()

Set the POSIX flags and the QNX Neutrino extended flags in a spawn attributes object

Synopsis:

```
#include <spawn.h>

int posix_spawnattr_setxflags(
    posix_spawnattr_t *attrp,
    uint32_t flags);
```

Arguments:

attrp

A pointer to the spawn attributes object that you want to modify.

flags

A bitwise OR of the flags you want to store in the spawn attributes.

POSIX defines the following flags:

- POSIX_SPAWN_SETPGROUP
- POSIX_SPAWN_SETSIGMASK
- POSIX_SPAWN_SETSIGDEF
- POSIX_SPAWN_SETSCHEDULER
- POSIX_SPAWN_SETSCHEDPARAM
- POSIX_SPAWN_RESETIDS

QNX Neutrino defines the following extended flags:

- POSIX_SPAWN_ALIGN_DEFAULT
- POSIX_SPAWN_ALIGN_FAULT
- POSIX_SPAWN_ALIGN_NOFAULT
- POSIX_SPAWN_ASLR_INVERT
- POSIX_SPAWN_CHECK_SCRIPT
- POSIX_SPAWN_DEBUG
- POSIX_SPAWN_EXEC
- POSIX_SPAWN_EXPLICIT_CPU
- POSIX_SPAWN_HOLD
- POSIX_SPAWN_NEWAPP
- POSIX_SPAWN_NOZOMBIE
- POSIX_SPAWN_PADDR64_SAFE

- `POSIX_SPAWN_SEARCH_PATH`
- `POSIX_SPAWN_SETCRED`
- `POSIX_SPAWN_SETND`
- `POSIX_SPAWN_SETSID`
- `POSIX_SPAWN_SETSIGIGN`
- `POSIX_SPAWN_SETSPART`
- `POSIX_SPAWN_SETSTACKMAX`
- `POSIX_SPAWN_TCSETPGROUP`

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_spawnattr_setxflags()` function stores the POSIX flags and QNX Neutrino extended flags in the spawn attributes object pointed to by *attrp*, overwriting any previously saved flags. You must have already initialized the spawn attributes object by calling [posix_spawnattr_init\(\)](#) (p. 2093).

These flags indicate which process attributes you want to change in the new process image when you invoke `posix_spawn()` or `posix_spawnnp()`. For detailed information about the flags, see the entry for [posix_spawn\(\)](#) (p. 2037).



The `posix_spawnattr_setxflags()` function is a QNX Neutrino extension. POSIX specifies the [posix_spawnattr_getflags\(\)](#) (p. 2068) and [posix_spawnattr_setflags\(\)](#) (p. 2098) routines, which get and set only the POSIX flags.

Returns:**EOK**

Success.

EINVAL

An argument was invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

posix_typed_mem_get_info()

Get information about a typed memory object

Synopsis:

```
#include <sys/mman.h>

int posix_typed_mem_get_info(
    int fildes,
    struct posix_typed_mem_info *info);
```

Arguments:

fildes

The file descriptor for the typed memory object that you want to query, returned by a call to [posix_typed_mem_open\(\)](#) (p. 2124).

info

A pointer to a `posix_typed_mem_info` structure where the function can store the information.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_typed_mem_get_info()` function queries the typed memory object to obtain the amount of memory currently available. This is particularly important for typed memory objects that may in some cases be scarce resources.



This function was added in the QNX Neutrino Core OS 6.3.2.

The `posix_typed_mem_get_info()` function stores, in the `posix_tmi_length` field of the `posix_typed_mem_info` structure pointed to by `info`, the maximum length that may be successfully allocated by the typed memory object designated by `fildes`.



If the typed memory pool is a shared resource, some form of mutual-exclusion or synchronization may be required while querying and allocating it, to prevent race conditions.

The maximum length is dynamic; it's valid only while the current mapping of the corresponding typed memory pool remains unchanged. It takes into account the `POSIX_TYPED_MEM_ALLOCATE` or `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag specified when the typed memory object was opened.

As a QNX Neutrino extension, if *fildev* represents a typed memory object opened with neither `POSIX_TYPED_MEM_ALLOCATE` nor `POSIX_TYPED_MEM_ALLOCATE_CONTIG`, the function sets *info->posix_tmi_length* to 0.

Returns:

0

Success.

EBADF

The *fildev* argument isn't a valid open file descriptor.

ENODEV

The *fildev* isn't connected to a memory object supported by this function.

Classification:

[POSIX 1003.1 TYM](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

posix_typed_mem_open()

Open a typed memory object

Synopsis:

```
#include <sys/mman.h>
#include <fcntl.h>

int posix_typed_mem_open( const char * name,
                          int oflag,
                          int tflag);
```

Arguments:

name

A pointer to the string that specifies the typed memory object.

oflag

The access mode to use; one of:

O_RDONLY

Open for read access only.

O_WRONLY

Open for write access only.

O_RDWR

Open for read or write access.

tflag

Flags that determine how the typed memory object behaves when mapped; at most one of:

POSIX_TYPED_MEM_ALLOCATE

Allocate on *mmap()*.

POSIX_TYPED_MEM_ALLOCATE_CONTIG

Allocate contiguously on *mmap()*.

POSIX_TYPED_MEM_MAP_ALLOCATABLE

Map on *mmap()* without affecting the availability for allocation.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_typed_mem_open()` function establishes a connection between the typed memory object specified by the *name* and a file descriptor. It creates an open file description that refers to the typed memory object and a file descriptor that refers to the open file description.



This function was added in the QNX Neutrino Core OS 6.3.2.

The names of typed memory regions are derived directly from the names of the *asinfo* segments. The *asinfo* section itself describes a hierarchy, and so the naming of typed memory object is a hierarchy.

The `posix_typed_mem_open()` function resolves the name as follows:

1. If the name starts with a leading `/`, an exact match is done.
2. The name may contain intermediate `/` characters. These are considered as path component separators. If multiple path components are specified, they're matched from the bottom up (the opposite of the way filenames are resolved).
3. If the name doesn't start with a leading `/`, a tail match is done on the pathname components specified.

For more information, see “Typed memory” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

The file descriptor is used by other functions to refer to that typed memory object.

The setting of *tflag* affects what happens when you call `mmap()` (p. 1656) with the file descriptor for the typed memory object:

POSIX_TYPED_MEM_ALLOCATE

The `mmap()` function allocates and maps typed memory from the specified typed memory pool. The allocated memory is either a previously unallocated contiguous area of the typed memory pool, or several noncontiguous previously unallocated areas (mapped to a contiguous portion of the process address space).

POSIX_TYPED_MEM_ALLOCATE_CONTIG

The *mmap()* function allocates and maps a previously unallocated single contiguous area of the typed memory pool (also mapped to a contiguous portion of the process address space).

**Neither `POSIX_TYPED_MEM_ALLOCATE` nor
`POSIX_TYPED_MEM_ALLOCATE_CONTIG`**

The *mmap()* function maps an application-chosen area from the specified typed memory pool.



The mapped area is unavailable for allocation until all processes unmap it.

`POSIX_TYPED_MEM_MAP_ALLOCATABLE`

The *mmap()* function maps an application-chosen area from the specified typed memory pool without affecting the availability of that area for allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping, or allocated if it was allocated prior to the mapping.

If *tflag* is 0 or `POSIX_TYPED_MEM_MAP_ALLOCATABLE`, the offset parameter to *mmap()* specifies the starting physical address in the typed memory region; if the typed memory region is discontinuous (multiple *asinfo* entries), the allowed offset values are also discontinuous and don't start at zero as they do for shared memory objects. If you specify a [*paddr*, *paddr* + *size*) region that falls outside the allowed addresses for the typed memory object, *mmap()* fails with `ENXIO`.

The *oflag* controls what permissions you're allowed to *mmap()* with. For example, if *oflag* is `O_RDONLY`, you can't do a *mmap()* with `MAP_SHARED` and `PROT_WRITE`.

If successful, *posix_typed_mem_open()* returns a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file descriptor is new and isn't shared with any other processes. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor is cleared.

The *mmap()* (p. 1656), *posix_mem_offset()* (p. 2030), *posix_typed_mem_get_info()* (p. 2122), *fstat()* (p. 779), *dup()* (p. 483), *dup2()* (p. 485), and *close()* (p. 336) functions can operate on a file descriptor that *posix_typed_mem_open()* returns.

As QNX Neutrino extensions:

- *msync()* (p. 1825) flushes and/or invalidates the data and or instruction cache for the specified memory region
- *ftruncate()* (p. 793) and other file operations indicate an error of `ENOSYS`

Returns:

A nonnegative integer that represents the lowest-numbered unused file descriptor, or -1 if an error occurred (*errno* is set).

Errors:**EACCES**

The typed memory object exists, and the permissions specified by *oflag* are denied.

EINTR

The *posix_typed_mem_open()* function was interrupted by a signal.

EINVAL

The flags specified in *tflag* are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEM_ALLOCATE_CONTIG, and POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the name argument exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENFILE

Too many file descriptors are currently open in the system.

ENOENT

The named typed memory object doesn't exist.

EPERM

The caller lacks the superuser capability to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in the *tflag* argument.

Classification:

POSIX 1003.1 TYM (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

pow(), powf(), powl()

Raise a number to a given power

Synopsis:

```
#include <math.h>

double pow( double x,
            double y );

float powf( float x,
            float y );

long double powl( long double x,
                  long double y );
```

Arguments:

x

The number you want to raise.

y

The power you want to raise the number to.

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The *pow()*, *powf()*, and *powl()* functions compute x raised to the power of y .

A domain error occurs if $x = 0$, and $y \neq 0$, or if x is negative, and y isn't an integer. A range error may also occur.

Returns:

The value of x^y . For finite values of x less than 0 and finite non-integer values of y , these functions return NaN. The return value for a correct value that would cause underflow is 0.0.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set *errno* (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", pow( 1.5, 2.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

2.755676

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

pread(), pread64()

Read from a file without moving the file pointer

Synopsis:

```
#include <unistd.h>

ssize_t pread(int filedes,
              void *buff,
              size_t nbytes,
              off_t offset );

ssize_t pread64( int filedes,
                 void *buff,
                 size_t nbytes,
                 off64_t offset );
```

Arguments:

filedes

The descriptor of the file that you want to read from.

buff

A pointer to a buffer where the function can store the data that it reads.

nbytes

The number of bytes that you want to read.

offset

The desired position inside the file.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pread()* and *pread64()* functions perform the same action as [read\(\)](#) (p. 2502), except that they read from a given position in the file without changing the file pointer. The *pread64()* function is a 64-bit version of *pread()*.

The *pread()* function reads up to the maximum offset value that can be represented in an *off_t* for regular files. An attempt to perform a *pread()* on a file that's incapable of seeking results in an error.

The *pread64()* function is a 64-bit version of *pread()*.

Returns:

The number of bytes actually read, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:**EAGAIN**

The *O_NONBLOCK* flag is set for the file descriptor, and the process would be delayed in the read operation.

EBADF

The file descriptor, *filides*, isn't a valid file descriptor open for reading.

EINTR

The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file does not report partial transfers.

EIO

A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSYS

The *pread()* function isn't implemented for the filesystem specified by *filides*.

Classification:

pread() is [POSIX 1003.1 XSI](#) (p. 46); *pread64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

printf()

Write formatted output to stdout

Synopsis:

```
#include <stdio.h>

int printf( const char * format,
            ... );
```

Arguments:

format

A string that controls the format of the output, as described below. The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *printf()* function writes output to the *stdout* stream, under control of the argument *format*.



If the format string contains invalid multibyte characters, processing stops, and the rest of the format string, including the % characters, is printed. This can happen, for example, if you specify international characters, accents, and diacritical marks using ISO 8859-1 instead of UTF-8. If you call:

```
setlocale( LC_CTYPE, "C-TRADITIONAL" );
```

before calling *printf()*, the locale switches multibyte processing from UTF-8 to 1-to-1, and *printf()* safely transfers the misformed multibyte characters.

Format Arguments

If there are leftover arguments after processing *format*, they're ignored.

The *printf()* family of functions allows for language-dependent radix characters. The default character is “.”, but is controlled by `LC_NUMERIC` and [setlocale\(\)](#) (p. 2880).

Format control string

The format control string consists of:

multibyte characters

These are copied to the output stream exactly as they occur in the *format* string. An ordinary character in the *format* string is any character, other than a percent character (%), that isn't part of a conversion specifier.

conversion specifiers

These cause argument values to be written as they're encountered during the processing of the *format* string. A conversion specifier is a sequence of characters in the *format* string that begins with “%” and is followed by:

- zero or more *format control flags* (p. 2134) that can modify the final effect of the format directive
- an optional decimal integer, or an asterisk (*), that specifies a *minimum field width* (p. 2135) to be reserved for the formatted item
- an optional *precision specification* (p. 2135) in the form of a period (.), followed by an optional decimal integer or an asterisk (*)
- an optional *type length* (p. 2135) specification, one of: h, hh, j, l, ll, L, t or z.
- a character that specifies the *type of conversion* (p. 2136) to be performed. See below.

Format control flags

The valid format control flags are:

-

Left-justify the formatted item within the field; normally, items are right-justified.

+

Always start a signed, positive object with a plus character (+); normally, only negative items begin with a sign.

space

Always start a signed, positive object with a space character; if both + and a space are specified, + overrides the space.

#

Use an alternate conversion form:

- For o (unsigned octal) conversions, increment the precision, if necessary, so that the first digit is 0.

- For x or X (unsigned hexadecimal) conversions, prepend a nonzero value with 0x or 0X.
- For e, E, f, g, or G (any floating-point) conversions, always include a decimal-point character in the result, even if no digits follow it; normally, a decimal-point character appears in the result only if there is a digit to follow it.
- In addition, for g or G conversions, don't remove trailing zeros from the result.

0 (zero)

Use leading zeros to pad the field width for d, i, o, u, x, X, e, E, f, g and G conversions. The “-” flag overrides the this flag.

Field width

If you don't specify a field width, or if the given value is less than the number of characters in the converted value (subject to any precision value), a field of sufficient width to contain the converted value is used.

If the converted value has fewer characters than specified by the field width, the value is padded on the left (or right, subject to the left-justification flag) with spaces or zero characters (0). If the field width begins with a zero, the value is padded with zeros; otherwise, the value is padded with spaces.

If the field width is * (or 0*), a value of type `int` from the argument list is used (before a precision argument or a conversion argument) as the minimum field width. A negative field width value is interpreted as a left-justification flag, followed by a positive field width.

Precision specifier

As with the field width specifier, a precision specifier of * causes a value of type `int` from the argument list to be used as the precision specifier.

The precision value affects the following conversions:

- For d, i, o, u, x and X (integer) conversions, the precision specifies the minimum number of digits to appear.
- For e, E and f (fixed-precision, floating-point) conversions, the precision specifies the number of digits to appear after the decimal-point character.
- For g and G (variable-precision, floating-point) conversions, the precision specifies the maximum number of significant digits to appear.
- For s (string) conversions, the precision specifies the maximum number of characters to appear.

Type length specifier

A type length specifier affects the conversion as follows:

- `h` causes a `d`, `i`, `o`, `u`, `x` or `X` (integer) format conversion to treat the argument as a `short` or `unsigned short` argument.

Note that, although the argument may have been promoted to an `int` as part of the function call, the value is converted to the smaller type before it's formatted.

- `h` causes an `n` (converted length assignment) operation to assign the converted length to an object of type `short`.
- `hh` is similar to `h`, but treats the argument as a `signed char` or an `unsigned char`.
- `j` causes a `d`, `i`, `o`, `u`, `x`, `X` format conversion to process a `intmax_t` or `uintmax_t`.
- `j` causes an `n` format conversion to process an `intmax_t`.
- `L` causes an `a`, `A`, `e`, `E`, `f`, `g`, `G` (double) format conversion to process a `long double` argument.
- `l` ("`el`") causes a `c` format conversion to process a `wint_t` argument.
- `l` ("`el`") causes an `s` format conversion to process a `wchar_t` argument.
- `l` ("`el`") causes a `d`, `i`, `o`, `u`, `x`, or `X` (integer) format conversion to process a `long` or `unsigned long` argument.
- `l` ("`el`") causes an `n` (converted length assignment) operation to assign the converted length to an object of type `long`.
- `ll` (double "`el`") causes a `d`, `i`, `o`, `u`, `x`, or `X` (integer) format conversion to assign the converted value to an object of type `long long` or `unsigned long long`.
- `ll` (double "`el`") causes an `n` (converted length assignment) operation to assign the number of characters that have been read to an object of type `long long`.
- `t` causes a `d`, `i`, `o`, `u`, `x`, `X` format conversion to process a `ptrdiff_t` or the corresponding `unsigned` type argument.
- `t` causes an `n` format conversion to process a `ptrdiff_t` argument.
- `z` causes a `d`, `i`, `o`, `u`, `x`, `X` format conversion to process a `size_t` argument.
- `z` causes an `n` format conversion to process a `size_t` argument.

Conversion type specifiers

The valid conversion type specifiers are:

a, A

Convert an argument of type `double` in the style `[-]0xh.hhhh p1d`, where there's one nonzero hexadecimal digit before the decimal point. The number of hexadecimal digits after the decimal point is equal to the precision. If the precision is missing and `FLT_RADIX` is a power of 2, then the precision is sufficient for an exact representation. If the precision is zero and you don't specify the `#` flag, no decimal point is shown.

The `a` conversion uses the letters `abcdef` and produces `x` and `p`; the `A` conversion `ABCDEF`, `X` and `P`. The exponent always has one digit, even if it's 0, and no more digits than necessary. The values for infinity or NaN are converted in the style of an `f` or `F`.

c

Convert an argument of type `int` into a value of type `unsigned char` and write the corresponding ASCII character code to the output stream.

An `l` (“`l`”) qualifier causes a `wint_t` argument to be converted as if by an `ls` conversion into a `wchar_t`, the first element being the `wint_t` and the second being a null wide character.

d, i

Convert an argument of type `int` into a signed decimal notation and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

e, E

Convert an argument of type `double` into a decimal notation in the form `[ - ]d.ddde[+ | - ]dd`. The leading sign appears (subject to the format control flags) only if the argument is negative.

If the argument is nonzero, the digit before the decimal-point character is nonzero. The precision is used as the number of digits following the decimal-point character. If you don't specify the precision, a default precision of six is used. If the precision is 0, the decimal-point character is suppressed. The value is rounded to the appropriate number of digits.

The exponent sign and the exponent (that indicates the power of ten by which the decimal fraction is multiplied) are always produced. The exponent is at least two digits long and has only as many additional digits as necessary to represent it. If the value is zero, the exponent is zero.

For `E` conversions, the exponent begins with the character `E`, rather than `e`.

The arguments infinity or NaN are converted in the style of the `f` or `F` conversion specifiers.

f, F

Convert an argument of type `double` into a decimal notation in the form `[ - ]ddd.ddd` with the number of digits after the decimal point being equal to the precision specification. The leading sign appears (subject to the format control flags) only if the argument is negative.

The precision is used as the number of digits following the decimal-point character. If you don't specify the precision, a default precision of six is used. If the precision is 0, the decimal-point character is suppressed; otherwise, at least one digit is produced before the decimal-point character. The value is rounded to the appropriate number of digits.

An argument of type `double` that represents infinity or NaN is converted to `[-]inf` or `[-]nan`. The `F` specifier produces `[-]INF` or `[-]NAN`.

g, G

Convert an argument of type `double` using either the `e` or `f` (or `E`, for a `G` conversion) style of conversion, depending on the value of the argument. In either case, the precision specifies the number of significant digits that are contained in the result. The `e` style conversion is used only if the exponent from such a conversion would be less than -4 or greater than the precision. Trailing zeros are removed from the result, and a decimal-point character only appears if it is followed by a digit.

Arguments representing infinity or NaN are converted in the style of the `f` or `F` conversion specifiers.

n

Assign the number of characters that have been written to the output stream to the integer pointed to by the argument. No output is produced.

o

Convert an argument of type `unsigned` into an unsigned octal notation, and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

p

Convert an argument of type `void *` into a value of type `int`, and format the value as for a hexadecimal (`x`) conversion.

s

Write the characters from the string specified by an argument of type `char *`, up to, but not including the terminating NUL character (`'\0'`), to the output stream. If you specify a precision, no more than that many characters are written.

If you use an `l` ("el") qualifier, the argument is interpreted as a pointer to a `wchar_t` array, and each wide character, including the terminating NUL, is converted as if by a call to [`wcrtomb\(\)`](#) (p. 3671). The terminating NUL is

written only if you don't specify the precision, or if you specify the precision and the length of the character sequence is less than the precision.

u

Convert an argument of type `unsigned` into an unsigned decimal notation, and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

x, X

Convert an argument of type `unsigned` into an unsigned hexadecimal notation, and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

Hexadecimal notation uses the digits 0 through 9 and the characters a through f or A through F for x or X conversions, respectively, as the hexadecimal digits. Subject to the alternate-form control flag, 0x or 0X is prepended to the output.

%

Print a % character (The entire specification is %%).

Any other conversion type specifier character, including another percent character (%), is written to the output stream with no special interpretation.

The arguments must correspond with the conversion type specifiers, left to right in the string; otherwise, indeterminate results will occur.

If the value corresponding to a floating-point specifier is infinity, or not a number (NaN), then the output will be `inf` or `-inf` for infinity, and `nan` or `-nan` for NaNs.

For example, a specifier of the form `%8.*f` defines a field to be at least 8 characters wide, and gets the next argument for the precision to be used in the conversion.

Returns:

The number of characters written, excluding the terminating NULL, or a negative number if an error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *weekday, *month;

    weekday = "Saturday";
    month = "April";
    printf( "%s, %s %d, %d\n", weekday, month, 10, 1999 );
    printf( "f1 = %8.4f f2 = %10.2E x = %#08x i = %d\n",
           23.45, 3141.5926, 0x1db, -1 );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Saturday, April 10, 1999  
f1 = 23.4500 f2 = 3.14E+003 x = 0x0001db i = -1
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

procmgr_ability()

Control a process's ability to perform certain operations

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_ability( pid_t pid,
                    unsigned ability, ... );
```

Arguments:

pid

The process ID of the process whose abilities you want to control, or 0 to control those of the calling process.



You need to be running as **root** in order to change a different process's abilities.

ability

A list of the abilities. Each ability in the list is composed of three separate components that are ORed together:

- An identifier (PROC_MGR_AID_*, or an custom ability found with [procmgr_ability_lookup\(\)](#) (p. 2154)) that tells procmgr to which particular ability you're modifying.
- One or more operations (PROC_MGR_AOP_*) that identify the operation you're performing on the ability.
- One or more domains (PROC_MGR_ADN_*) that indicate whether you're modifying what you can do with the ability while running as root or non-root.

Terminate the list with PROC_MGR_AID_EOL.

lower, upper

If the ability is for a subrange (see below), follow it by two uint64_t arguments that specify the lower and upper limits on the range.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `procmgr_ability()` function takes a list of ability operations to control what the identified process is allowed to do. This function lets a process start running as `root`, set the abilities it needs, and then change its group and user IDs. This can make your system more secure by reducing the number of potentially vulnerable processes running as `root`.

The table below describes the identifier portion for each ability, indicates whether the operation is normally privileged (e.g. rebooting the system) or not (e.g. spawning and forking), and describes the subrange if applicable. If you want to specify a subrange, include `PROC_MGR_AOP_SUBRANGE` in the operation flags. You can also specify custom abilities that a server has defined by calling [procmgr_ability_create\(\)](#) (p. 2152).



If you want to set more than one ability, you must specify a separate identifier-operation-domain argument (with any subranges) for each. The identifiers are an enumeration; don't try to OR them together.

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
PROC_MGR_AID_ABLE_CREATE	Yes	Create additional abilities; for more information, see procmgr_ability_create() (p. 2152)	—
PROC_MGR_AID_APS_ROOT	Yes	Execute adaptive partitioning scheduler operations that usually require <code>root</code> permissions; for more information, see SchedCtl() (p. 2745)	—
PROC_MGR_AID_CHILD_NEWAPP	Yes	Create a new application ID for a child process by setting <code>POSIX_SPAWN_NEWAPP</code> for posix_spawn() or posix_spawnp() (p. 2037), or <code>SPAWN_NEWAPP</code> for the spawn*() (p. 3089) functions	—
PROC_MGR_AID_CLOCKPERIOD	Yes	Change the clock period, using ClockPeriod() (p. 329)	Allowable periods, in nanoseconds

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
PROC_MGR_AID_CLOCKSET	Yes	Set the clock, using clock_settime() (p. 319), settimeofday() (p. 2915), ClockAdjust() (p. 322), or ClockTime() (p. 333)	Allowable times, in nanoseconds
PROC_MGR_AID_CONFSET	Yes	Set configuration strings, using confstr() (p. 344)	Allowable names (<code>_CS_*</code>)
PROC_MGR_AID_CONNECTION	Yes	- Use ConnectFlags() (p. 369) to change the connection flags on a coid for another process with a different user ID - Use MsgSendPulse() (p. 1789) to send a pulse to a channel that was created by a different process with a different user ID - Use SyncMutexEvent() (p. 3291) with a <code>SIGEV_PULSE</code> sigevent that's going to a channel that was created by a different process with a different user ID	—
PROC_MGR_AID_CPUMODE	Yes	Change the CPU's power management mode	Allowable modes
PROC_MGR_AID_DEFAULT_TIMER_TOLERANCE	Yes	Set the default timer tolerance for a process, using procmgr_timer_tolerance() (p. 2178)	—
PROC_MGR_AID_EVENT	Yes	Trigger privileged system-wide events, using procmgr_event_trigger() (p. 2170)	Trigger bits
PROC_MGR_AID_FORK	No	Create a new process by calling fork() (p. 685)	—
PROC_MGR_AID_GETID	Yes	Get the group ID or session ID of a process outside the calling process's session, by using getpgid() (p. 917) or getsid() (p. 958), respectively	—
PROC_MGR_AID_INTERRUPT	Yes	Attach interrupt handlers by calling InterruptAttach() (p. 1147), or events by calling InterruptAttachEvent() (p. 1154).	Interrupt sources

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
		You also need I/O privileges; see PROC_MGR_AID_IO, below.	
PROC_MGR_AID_IO	Yes	Request I/O privileges by calling ThreadCtl (p. 3408)(<code>_NTO_TCTL_IO</code> , 0) or ThreadCtl (<code>_NTO_TCTL_IO_PRIV</code> , 0)	—
PROC_MGR_AID_KEYDATA	Yes	Pass data through a common client, by calling MsgKeyData() (p. 1747). Resource managers that don't run as root should retain this ability, or else DUP messages will fail.	—
PROC_MGR_AID_MAP_FIXED	No	Use mmap() (p. 1656) with <code>MAP_FIXED</code> to map fixed addresses (including zero)	Allowable virtual addresses
PROC_MGR_AID_MEM_ADD	Yes	Add physical memory	Allowable physical addresses
PROC_MGR_AID_MEM_GLOBAL	Yes	Mark shared memory as being global across all processes, by calling shm_ctl() (p. 2925) or shm_ctl_special() (p. 2931), specifying <code>SHMCTL_GLOBAL</code>	—
PROC_MGR_AID_MEM_LOCK	Yes	Lock a range of process address space into physical memory, by calling mlock() (p. 1651) or mlockall() (p. 1653)	Allowable virtual addresses
PROC_MGR_AID_MEM_PEER	Yes	Manipulate a peer process's memory	Peer process IDs
PROC_MGR_AID_MEM_PHYS	Yes	- Map physical memory, by calling mmap() (p. 1656), specifying <code>MAP_PHYS</code>, or by calling mmap_device_io() (p. 1666) or mmap_device_memory() (p. 1668) - Use <code>SHMCTL_PHYS</code> with shm_ctl() (p. 2925) or shm_ctl_special() (p. 2931)	The lower and upper physical address range
PROC_MGR_AID_MEM_SPECIAL	Yes	Call shm_ctl_special() (p. 2931)	—

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
PROC_MGR_AID_PATHSPACE	Yes	Add items to the <code>procnto</code> pathname prefix space, specifically to create symbolic links by calling pathmgr_symlink() (p. 1943), or to register names in the path space by calling resmgr_attach() (p. 2580)	—
PROC_MGR_AID_PATH_TRUST	Yes	Modify the list of trusted files and filesystems. For more information, see the entry for <code>pathtrust</code> in the <i>Utilities Reference</i> , as well as the description of <code>PROT_EXEC</code> in the entry for mmap() (p. 1656).	—
PROC_MGR_AID_PGRP	No	Set its process group ID, by calling setpgrp() (p. 2889) or procmgr_session() (p. 2175). This ability is enabled by default (for POSIX conformance). You can disable it completely or restrict it to specific pid ranges.	Process IDs
PROC_MGR_AID_PRIORITY	Yes	- Use pthread_setschedparam() (p. 2400), pthread_setschedprio() (p. 2402), sched_setparam() (p. 2738), sched_setscheduler() (p. 2740), setprio() (p. 2890), or SchedSet() (p. 2780) to change its priority to one that's above the maximum allowed for unprivileged processes - Use pthread_mutexattr_setprioceiling() (p. 2349), pthread_mutexattr_setprotocol() (p. 2351), pthread_mutex_setprioceiling() (p. 2322), SyncCtl() (p. 3286), or SyncTypeCreate() (p. 3305) to set the priority ceiling for a mutex to a value greater than the maximum allowed for unprivileged processes The maximum is usually 63, but is governed by the <code>-P</code> option to <code>procnto</code>.	Allowable priorities

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
PROC_MGR_AID_PROT_EXEC	No	Load code by calling dlopen() (p. 454) or map memory as executable by calling mmap() (p. 1656) with PROT_EXEC	Allowable virtual addresses
PROC_MGR_AID_PUBLIC_CHANNEL	No	Create a public channel by calling ChannelCreate() (p. 278) without setting _NTO_CHF_PRIVATE	—
PROC_MGR_AID_QNET	Yes	Used by Qnet when it creates a channel.	—
PROC_MGR_AID_RCONSTRAINT	Yes	Operate without any resource constraints. For more information, see Resource constraint thresholds in the Processes chapter of the QNX Neutrino <i>Programmer's Guide</i> .	—
PROC_MGR_AID_REBOOT	Yes	Cause the system to reboot by calling sysmgr_reboot() (p. 3319)	—
PROC_MGR_AID_RLIMIT	Yes	Use setrlimit() (p. 2899) to raise hard limits on system resources	Limits (RLIMIT_*) that it can raise
PROC_MGR_AID_RSRCDBMGR	Yes	Use the rsrddbmgr*() (p. 2479) functions to manipulate the resource database manager	—
PROC_MGR_AID_RUNSTATE	Yes	Use sysmgr_runstate() (p. 3320) and sysmgr_runstate_dynamic() (p. 3320) to control a CPU's running state	Allowable CPU numbers
PROC_MGR_AID_RUNSTATE_BURST	No	Use sysmgr_runstate_burst() (p. 3322) to tell the kernel to turn on any offline CPUs because the system is about to get busy	The maximum length of time, in milliseconds, for which the process is allowed to set burst mode
PROC_MGR_AID_SCHEDULE	Yes	Use pthread_getschedparam() (p. 2299), SchedGet() (p. 2774),	—

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
		pthread_setschedparam() (p. 2400), or SchedSet() (p. 2780) to get or set the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID	
PROCmgr_AID_SESSION	Yes	Use procmgr_session() (p. 2175) to change a character terminal's process group or to send a signal to a member of a session group	Allowable session IDs
PROCmgr_AID_SETGID	Yes	Set its group ID, effective group ID, real and effective group IDs, or supplementary group IDs by calling setgid() (p. 2858), setegid() (p. 2851), setregid() (p. 2895), or setgroups() (p. 2864), respectively, or delete existing groups using setgroups()	Allowable group IDs
PROCmgr_AID_SETUID	Yes	- Use setuid() (p. 2917) to set its user ID and effective user ID to values other than its real or saved-set user ID - Use seteuid() (p. 2856) to set its effective user ID to a value other than its real or saved-set user ID - Use setreuid() (p. 2897) to set its user ID to a value other than its effective user ID, and to set its effective user ID to values other than its effective or saved-set user ID	Allowable user IDs
PROCmgr_AID_SIGNAL	Yes	- Attach signal handlers to a process with a different real or effective user ID by calling sigaction() (p. 2948), signal() (p. 2974), or SignalAction() (p. 2977) - Set the signal-blocked mask for a process with a different real or effective user ID by calling pthread_sigmask() (p. 2407),	Allowable signals

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
		sigprocmask() (p. 3004), or SignalProcmask() (p. 2991) Send signals to a process with a different real or effective user ID by calling kill() (p. 1494), pthread_kill() (p. 2310), sigqueue() (p. 3007), or SignalKill() (p. 2986)	
PROCGR_AID_SPAWN	No	Spawn new processes by calling exec*() (p. 487), spawn* (p. 2699), or posix_spawn() (p. 2037)	—
PROCGR_AID_SPAWN_SETGID	Yes	Set the group ID of the child process when using posix_spawn() (p. 2037)	Lower and upper bounds on the group IDs that the process can set the child process to
PROCGR_AID_SPAWN_SETUID	Yes	Set the user ID of the child process when using posix_spawn() (p. 2037)	Lower and upper bounds on the user IDs that the process can set the child process to
PROCGR_AID_SWAP	Yes	Enable, disable, or configure the memory swapper, pager	—
PROCGR_AID_TIMER	Yes	- Create a timer that sends a pulse by calling timer_create() (p. 3427) or TimerCreate() (p. 3451) - Get timer information by calling timer_getexpstatus() (p. 3433), timer_getoverrun() (p. 3435), timer_gettime() (p. 3437), or TimerInfo() (p. 3456)	Timer IDs

Identifier	Privileged?	Controls the process's ability to:	Subrange (optional)
PROC_MGR_AID_TRACE	Yes	Add handlers for trace events or allocate the instrumented kernel's trace buffers by calling TraceEvent() (p. 3510)	—
PROC_MGR_AID_UMASK	Yes	Use umask() (p. 3557) to change the file-mode creation mask for a process with a different effective user ID	—
PROC_MGR_AID_V86	Yes	Call _intr_v86() (p. 1185)	—
PROC_MGR_AID_WAIT	Yes	Use wait() (p. 3652), wait3() (p. 3655), wait4() (p. 3658), waitid() (p. 3663), or waitpid() (p. 3666) to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID	Child process IDs

You must OR in at least one of the following operations:

PROC_MGR_AOP_DENY

Disallow the performance of the operation in the specified domain(s).

PROC_MGR_AOP_ALLOW

Allow the performance of the operation in the specified domain(s). You must be `root` when doing this for privileged abilities.

PROC_MGR_AOP_SUBRANGE

Restrict the feature to set its “parameter” to a certain subrange in the specified domain(s). You must still have the `PROC_MGR_AOP_ALLOW` flag set for the ability in the domain to successfully use the feature.

The meaning of the “parameter” varies, depends on on the ability, as described above.

Follow the ability entry in the function's parameter list with two `uint64_t` arguments that give the lower and upper bounds of the subrange. You must be `root` when doing this for privileged abilities.

You can have multiple subrange entries in the list; they “OR” together to let you form a discontinuous set. To do this, specify the identifier-operation-domain argument again, followed by another pair of range arguments.

PROC_MGR_AOP_LOCK

Lock the current ability so that no further changes to it can be made.

PROC_MGR_AOP_INHERIT_YES

The changes to the ability are inherited across a *spawn* or *exec*.

PROC_MGR_AOP_INHERIT_NO

The changes to the ability aren't inherited across a *spawn* or *exec*. This is the default.



For a *fork()* (p. 685), the parent's ability configuration is copied to the child verbatim, regardless of the INHERIT status of each of the abilities.

You must OR in at least one of the following for the domain portion:

PROC_MGR_ADN_ROOT

Modify the ability of the process when it's running as root.

PROC_MGR_ADN_NONROOT

Modify the ability of the process when it isn't running as root.

There's also a special ability identifier, **PROC_MGR_AID_EOL**, that indicates the end of the list of abilities passed to this function.

You can OR the domain flags and any of **PROC_MGR_AOP_DENY**, **PROC_MGR_AOP_ALLOW**, and **PROC_MGR_AOP_LOCK** into the **PROC_MGR_AID_EOL** macro that ends the list. If you do this, the operations are performed on all the unlocked abilities that you *didn't* specify in the list.

Returns:**EOK**

The abilities were successfully modified.

EINVAL

There's something wrong with the parameters to the function.

EPERM

A non-root process tried to give itself a privileged ability or tried to change a locked ability.

Examples:

Remove the ability for a root process to set the user ID when spawning a process:

```
procmgr_ability(0,
```

```
PROCMGR_ADN_ROOT|PROCMGR_AOP_DENY|PROCMGR_AID_SPAWN_SETUID,
PROCMGR_AID_EOL);
```

Regain the ability for a root process to set the user ID:

```
procmgr_ability(0,
PROCMGR_ADN_ROOT|PROCMGR_AOP_ALLOW|PROCMGR_AID_SPAWN_SETUID,
PROCMGR_AID_EOL);
```

Drop all abilities while running as root and lock out any further changes:

```
procmgr_ability(0,
PROCMGR_ADN_ROOT|PROCMGR_AOP_DENY|PROCMGR_AOP_LOCK|PROCMGR_AID_EOL);
```

Allow a non-root process to set the user ID while spawning, but restrict the number to 10000 or higher and lock out any further changes. Remove all other abilities while running as root, and lock any further changes to them as well:

```
procmgr_ability(0,
PROCMGR_ADN_NONROOT|PROCMGR_AOP_ALLOW|PROCMGR_AID_SPAWN_SETUID,
PROCMGR_ADN_NONROOT|PROCMGR_AOP_SUBRANGE|PROCMGR_AOP_LOCK|PROCMGR_AID_SPAWN_SETUID,
(uint64_t)10000, ~(uint64_t)0,
PROCMGR_ADN_ROOT|PROCMGR_AOP_DENY|PROCMGR_AOP_LOCK|PROCMGR_AID_EOL);
```

Do the same as the above, but specify a discontinuous set of user IDs by using a second PROCMGR_AID_SPAWN_SETUID ability:

```
procmgr_ability(0,
PROCMGR_ADN_NONROOT|PROCMGR_AOP_ALLOW|PROCMGR_AID_SPAWN_SETUID,
PROCMGR_ADN_NONROOT|PROCMGR_AOP_SUBRANGE|PROCMGR_AID_SPAWN_SETUID,
(uint64_t)1000, (uint64_t)1050,
PROCMGR_ADN_NONROOT|PROCMGR_AOP_SUBRANGE|PROCMGR_AOP_LOCK|PROCMGR_AID_SPAWN_SETUID,
(uint64_t)2000, (uint64_t)2013,
PROCMGR_ADN_ROOT|PROCMGR_AOP_DENY|PROCMGR_AOP_LOCK|PROCMGR_AID_EOL);
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_ability_create()

Create a process-manager ability

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_ability_create( const char *name,
                           unsigned flags );
```

Arguments:

name

The name of the ability. This should by convention consist of a service identifier followed by a capability identifier (e.g., "fs-qnx6/some_devctl").

flags

Flags that apply to the new ability; a bitwise OR of zero or more of the following:

- PROC_MGR_ADN_ROOT — the ability applies to privileged processes by default.
- PROC_MGR_ADN_NONROOT — the ability applies to unprivileged processes by default.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *procmgr_ability_create()* function creates a process-manager ability. A server can use this function to create the ability, and clients can use [*procmgr_ability_lookup\(\)*](#) (p. 2154) to look it up. The *name* argument is a string that uniquely identifies the ability, and the *flags* specify which privilege domains have the ability by default.

- In order to create an ability, your process must have the PROC_MGR_AID_ABLE_CREATE ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- You can only set the privilege domains for a new ability once; additional calls to *procmgr_ability_create()* for the same ability will fail unless the flags are the same as before.
- There's no requirement for a call to *procmgr_ability_create()* to precede calls to *procmgr_ability_lookup()*. This avoids forcing any specific ordering of process initialization.



Returns:

A numeric ability identifier, which can then be used in a call to *procmgr_ability()* or to verify the abilities of a client, or a negative [errno](#) (p. 539) value if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_ability_lookup()

Look up a process-manager ability

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_ability_lookup( const char *name );
```

Arguments:

name

The name of the ability. This should by convention consist of a service identifier followed by a capability identifier (e.g., "fs-qnx6/some_devctl").

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *procmgr_ability_lookup()* function looks up a process-manager ability that was (or will be) created by a call to [procmgr_ability_create\(\)](#) (p. 2152).



There's no requirement for a call to *procmgr_ability_create()* to precede calls to *procmgr_ability_lookup()*. This avoids forcing any specific ordering of process initialization.

Returns:

A numeric ability identifier, which can then be used in a call to *procmgr_ability()* or to verify the abilities of a client, or a negative [errno](#) (p. 539) value if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_daemon()

Run a process in the background

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_daemon( int status,
                   unsigned flags );
```

Arguments:

status

The status that you want to return to the parent process.

flags

The flags currently defined (in `<sys/procmgr.h>`) are:

- `PROCMGR_DAEMON_NOCHDIR` — unless this flag is set, *procmgr_daemon()* changes the current working directory to the root “/”.
- `PROCMGR_DAEMON_NOCLOSE` — unless this flag is set, *procmgr_daemon()* closes all file descriptors other than standard input, standard output and standard error.
- `PROCMGR_DAEMON_NODEVNULL` — unless this flag is set, *procmgr_daemon()* redirects standard input, standard output and standard error to `/dev/null`.
- `PROCMGR_DAEMON_KEEPMASK` — unless this flag is set, *procmgr_daemon()* sets the umask to 0 (zero).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *procmgr_daemon()* function lets programs detach themselves from the controlling terminal and run in the background as system daemons. This also puts the caller into session 1.

The High Availability Manager (HAM) automatically monitors processes detached using `procmgr_daemon()` because they're in session 1. For more information, see the High Availability Framework *Developer's Guide*.

The argument `status` is returned to the parent process as if `exit()` (p. 599) were called; the returned value is normally `EXIT_SUCCESS`.



The data in the `siginfo_t` structure for the `SIGCHLD` signal that the parent receives isn't useful in this case.

Returns:

A nonnegative integer, or -1 if an error occurs.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_event_notify()

Ask to be notified of system-wide events

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_event_notify (
    unsigned flags,
    const struct sigevent * event );
```

Arguments:

flags

A bitwise OR of the type of events that you want to be notified of, or 0 to unarm the sigevent. The event types include:

- PROCmgr_EVENT_CONFSTR
- PROCmgr_EVENT_DAEMON_DEATH
- PROCmgr_EVENT_PATHSPACE
- PROCmgr_EVENT_SYNC
- PROCmgr_EVENT_SYSCONF
- PROCmgr_EVENT_TOD

For more information, see “[Event types](#) (p. 2159),” below.

event

A pointer to a [sigevent](#) (p. 2961) structure that specifies how you want to be notified.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *procmgr_event_notify()* function requests that the process manager notify the caller of the system-wide events identified by the given *flags*.



This function lets you add a single notification request for the process. If you call it again, the new request replaces the previous one. If you want to your process to have more than one notification request, use [procmgr_event_notify_add\(\)](#) (p. 2163) and [procmgr_event_notify_delete\(\)](#) (p. 2168) instead.

To remove the current notification request, pass a value of 0 for *flags*.

Event types

The following event types are defined in `<sys/procmgr.h>`:

PROC_MGR_EVENT_CONFSTR

A process set a configuration string.

PROC_MGR_EVENT_DAEMON_DEATH

A process in session 1 died. This event is most useful for watching for the death of daemon processes that use [procmgr_daemon\(\)](#) (p. 2156) to put themselves in session 1 as well as close and redirect file descriptors. As a result of this closing and redirecting, the death of daemons is difficult to detect otherwise.



Notification is via the given event, so no information is provided as to which process died. Once you've received the event, you'll need to do something else to find out if processes you care about had died. You can do this by walking through the list of all processes, looking for specific process IDs or process names. If you don't find one, then it has died. The sample code below demonstrates how you can do this.

PROC_MGR_EVENT_PATHSPACE

(QNX Momentics 6.3.0 SP2 or later) A resource manager added or removed an entry (i.e. mountpoint) to or from the pathname space. This is generally associated with resource manager calls to [resmgr_attach\(\)](#) (p. 2580) and [resmgr_detach\(\)](#) (p. 2600). Terminating a resource manager process also generates this event if the mountpoints haven't been detached.

PROC_MGR_EVENT_SYNC

A process called [sync\(\)](#) (p. 3279) to synchronize the filesystems.

PROC_MGR_EVENT_SYSCONF

A process set a system configuration string.

PROC_MGR_EVENT_TOD

(QNX Neutrino 6.6 or later) A process changed the time of day by calling *ClockTime()* (p. 333) or *clock_settime()* (p. 319).

Returns:

-1 on error; any other value indicates success.

Examples:

```
/*
 * This demonstrates procmgr_event_notify() with the
 * PROCMGR_DAEMON_DEATH flag. This flag allows you to
 * be notified if any process in session 1 dies.
 * Daemons are processes that do things that make
 * their death hard to detect (they become daemons by calling
 * procmgr_daemon()). One of the things that happens is that
 * daemons end up in session 1. Hence, the usefulness of the
 * PROCMGR_DAEMON_DEATH flag.
 *
 * When you are notified, you're not told who died.
 * It's up to you to know who should be running. Once notified,
 * you could then walk through the list of which processes are
 * still running and see if all the expected processes are still
 * running. If you know the process id of the processes you
 * are watching out for then this is easiest. If you don't know
 * the process id then your next option may be by process name.
 * The code below does a lookup by process name.
 */

#include <devctl.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <sys/procfs.h>
#include <sys/procmgr.h>

static int check_if_running( char *process );

#define DAEMON_DIED_CODE    (_PULSE_CODE_MINAVAIL)

struct dinfo_s {
    procfs_debuginfo    info;
    char                pathbuffer[PATH_MAX];
};

int
main( int argc, char **argv )
{
    char                *daemon_to_watch;
    int                 chid, coid, rcvid;
    struct sigevent event;
    struct _pulse       msg;

    if (argc != 2) {
        printf( "use: %s process_to_watch_for\n", argv[0] );
        exit( EXIT_FAILURE );
    }

    daemon_to_watch = argv[1];    /* the process to watch for */

    chid = ChannelCreate( 0 );
    coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );
    SIGEV_PULSE_INIT( &event, coid, SIGEV_PULSE_PRIO_INHERIT,
                      DAEMON_DIED_CODE, 0 );

    /*
     * Ask to be notified via a pulse whenever a
     * daemon process dies
     */
}
```

```

*/
if (procmgr_event_notify( PROCMGR_EVENT_DAEMON_DEATH,
                          &event ) == -1) {
    fprintf( stderr, "procmgr_event_notify() failed" );
    exit( EXIT_FAILURE );
}

while (1) {
    rcvid = MsgReceive( chid, &msg, sizeof(msg), NULL );
    if (rcvid != 0) {
        /* not a pulse; could be an unexpected message or
           error */
        exit( EXIT_FAILURE );
    }

    if (check_if_running( daemon_to_watch ) == 0)
        printf( "%s is no longer running\n", daemon_to_watch);
}
return 0;
}

/*
 * check_if_running - This will walk through all processes
 * to see if this particular one is still running.
 */

static int
check_if_running( char *process )
{
    DIR          *dirp;
    struct dirent *dire;
    char         buffer[20];
    int          fd, status;
    pid_t        pid;
    struct dinfo_s dinfo;

    if ((dirp = opendir( "/proc" )) == NULL) {
        perror( "Could not open '/proc'" );
        return -1;
    }
    while (1) {
        if ((dire = readdir( dirp )) == NULL)
            break;
        if (isdigit( dire->d_name[0] )) {
            pid = strtoul( dire->d_name, NULL, 0 );

            sprintf( buffer, "/proc/%d/as", pid );
            if ((fd = open( buffer, O_RDONLY )) != NULL) {
                status = devctl( fd, DCMD_PROC_MAPDEBUG_BASE,
                                &dinfo, sizeof(dinfo), NULL );
                if (status == EOK) {
                    if (!strcmp( process,
                                basename( dinfo.info.path ) ))
                        {
                            closedir (dirp);

                            /* You should close fd to prevent memory
                               leaking */
                            close(fd);

                            return 1;
                        }
                }
                /* else some errors are expected, e.g. procnto
                   has no MAPDEBUG info and there is a timing
                   issue with getting info on the process
                   that died, ignore errors */
                close( fd );
            }
        }
    }
    closedir( dirp );
    return 0;
}

```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_event_notify_add()

Add a request to be notified of system-wide events

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_event_notify_add (
    unsigned flags,
    const struct sigevent * event );
```

Arguments:

flags

A bitwise OR of the type of events that you want to be notified of. The event types include:

- PROCmgr_EVENT_CONFSTR
- PROCmgr_EVENT_DAEMON_DEATH
- PROCmgr_EVENT_PATHSPACE
- PROCmgr_EVENT_SYNC
- PROCmgr_EVENT_SYSCONF
- PROCmgr_EVENT_TOD

For more information, see “[Event types](#) (p. 2164),” below.

event

A pointer to a [sigevent](#) (p. 2961) structure that specifies how you want to be notified.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *procmgr_event_notify_add()* function adds a request that the process manager notify the caller of the system-wide events identified by the given *flags*. You can use this function to set up more than one notification request.

To delete a request, call [procmgr_event_notify_delete\(\)](#) (p. 2168), passing it the handle returned by [procmgr_event_notify_add\(\)](#).

Event types

The following event types are defined in `<sys/procmgr.h>`:

PROC_MGR_EVENT_CONFSTR

A process set a configuration string.

PROC_MGR_EVENT_DAEMON_DEATH

A process in session 1 died. This event is most useful for watching for the death of daemon processes that use [procmgr_daemon\(\)](#) (p. 2156) to put themselves in session 1 as well as close and redirect file descriptors. As a result of this closing and redirecting, the death of daemons is difficult to detect otherwise.



Notification is via the given event, so no information is provided as to which process died. Once you've received the event, you'll need to do something else to find out if processes you care about had died. You can do this by walking through the list of all processes, looking for specific process IDs or process names. If you don't find one, then it has died. The sample code below demonstrates how you can do this.

PROC_MGR_EVENT_PATHSPACE

(QNX Momentics 6.3.0 SP2 or later) A resource manager added or removed an entry (i.e. mountpoint) to or from the pathname space. This is generally associated with resource manager calls to [resmgr_attach\(\)](#) (p. 2580) and [resmgr_detach\(\)](#) (p. 2600). Terminating a resource manager process also generates this event if the mountpoints haven't been detached.

PROC_MGR_EVENT_SYNC

A process called [sync\(\)](#) (p. 3279) to synchronize the filesystems.

PROC_MGR_EVENT_SYSCONF

A process set a system configuration string.

PROC_MGR_EVENT_TOD

(QNX Neutrino 6.6 or later) A process changed the time of day by calling [ClockTime\(\)](#) (p. 333) or [clock_settime\(\)](#) (p. 319).

Returns:

An integer handle that you can pass to *procmgr_event_notify_delete()*, or -1 if an error occurred (*errno* (p. 539) is set).

Examples:

```

/*
 * This demonstrates procmgr_event_notify_add() with the
 * PROCMGR_DAEMON_DEATH flag. This flag allows you to
 * be notified if any process in session 1 dies.
 * Daemons are processes that do things that make
 * their death hard to detect (they become daemons by calling
 * procmgr_daemon()). One of the things that happens is that
 * daemons end up in session 1. Hence, the usefulness of the
 * PROCMGR_DAEMON_DEATH flag.
 *
 * When you are notified, you're not told who died.
 * It's up to you to know who should be running. Once notified,
 * you could then walk through the list of which processes are
 * still running and see if all the expected processes are still
 * running. If you know the process id of the processes you
 * are watching out for then this is easiest. If you don't know
 * the process id then your next option may be by process name.
 * The code below does a lookup by process name.
 */

#include <devctl.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <sys/procfs.h>
#include <sys/procmgr.h>

static int check_if_running( char *process );

#define DAEMON_DIED_CODE    (_PULSE_CODE_MINAVAIL)

struct dinfo_s {
    procfs_debuginfo    info;
    char                pathbuffer[PATH_MAX];
};

int
main( int argc, char **argv )
{
    char                *daemon_to_watch;
    int                 chid, coid, rcvid, handle;
    struct sigevent event;
    struct _pulse      msg;

    if (argc != 2) {
        printf( "use: %s process_to_watch_for\n", argv[0] );
        exit( EXIT_FAILURE );
    }

    daemon_to_watch = argv[1];    /* the process to watch for */

    chid = ChannelCreate( 0 );
    coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );
    SIGEV_PULSE_INIT( &event, coid, SIGEV_PULSE_PRIO_INHERIT,
                     DAEMON_DIED_CODE, 0 );

    /*
     * Ask to be notified via a pulse whenever a
     * daemon process dies
     */

    handle = procmgr_event_notify_add( PROCMGR_EVENT_DAEMON_DEATH, &event );
    if (handle == -1) {

```

```

        fprintf( stderr, "procmgr_event_notify_add() failed" );
        exit( EXIT_FAILURE );
    }

    while (1) {
        rcvid = MsgReceive( chid, &msg, sizeof(msg), NULL );
        if (rcvid != 0) {
            /* not a pulse; could be an unexpected message or
               error */
            exit( EXIT_FAILURE );
        }

        if (check_if_running( daemon_to_watch ) == 0)
            printf( "%s is no longer running\n", daemon_to_watch);
    }

    procmgr_event_notify_delete (handle);

    return 0;
}

/*
 * check_if_running - This will walk through all processes
 * to see if this particular one is still running.
 */

static int
check_if_running( char *process )
{
    DIR          *dirp;
    struct dirent *dire;
    char         buffer[20];
    int          fd, status;
    pid_t        pid;
    struct dinfo_s dinfo;

    if ((dirp = opendir( "/proc" )) == NULL) {
        perror( "Could not open '/proc'" );
        return -1;
    }

    while (1) {
        if ((dire = readdir( dirp )) == NULL)
            break;
        if (isdigit( dire->d_name[0] )) {
            pid = strtoul( dire->d_name, NULL, 0 );

            sprintf( buffer, "/proc/%d/as", pid );
            if ((fd = open( buffer, O_RDONLY )) != NULL) {
                status = devctl( fd, DCMD_PROC_MAPDEBUG_BASE,
                                &dinfo, sizeof(dinfo), NULL );
                if (status == EOK) {
                    if (!strcmp( process,
                                basename( dinfo.info.path ) ))
                    {
                        closedir (dirp);

                        /* You should close fd to prevent memory
                           leaking */
                        close(fd);

                        return 1;
                    }
                }
            } /* else some errors are expected, e.g. procnto
               has no MAPDEBUG info and there is a timing
               issue with getting info on the process
               that died, ignore errors */
            close( fd );
        }
    }

    closedir( dirp );
    return 0;
}

```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_event_notify_delete()

Delete a request to be notified of system-wide events

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_event_notify_delete ( const int handle );
```

Arguments:

handle

The handle for the request, returned by [procmgr_event_notify_add\(\)](#) (p. 2163) or [procmgr_value_notify_add\(\)](#) (p. 2181).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *procmgr_event_notify_delete()* function deletes a notification request that was previously added by a call to [procmgr_event_notify_add\(\)](#) (p. 2163). The *handle* is the value that *procmgr_event_notify_add()* returned.

Returns:

EOK

The notification request was successfully deleted.

ESRCH

The notification request with the given handle wasn't found.

Examples:

See the entry for [procmgr_event_notify_add\(\)](#) (p. 2163).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_event_trigger()

Trigger a system-wide event

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_event_trigger( unsigned flags );
```

Arguments:

flags

A bitwise OR of the types of event that you want to trigger:

- PROCMGR_EVENT_CONFSTR (privileged)
- PROCMGR_EVENT_DAEMON_DEATH (privileged)
- PROCMGR_EVENT_PATHSPACE
- PROCMGR_EVENT_SYNC
- PROCMGR_EVENT_SYSCONF (privileged)
- PROCMGR_EVENT_TOD

For descriptions of the events, see “[Event types](#) (p. 2164)” in the entry for [procmgr_event_notify_add\(\)](#) (p. 2158).



In order to trigger a privileged event, your process must have the PROCMGR_AID_EVENT ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *procmgr_event_trigger()* triggers a system-wide event. The event is sent to all processes that requested (via [procmgr_event_notify_add\(\)](#) (p. 2158) or [procmgr_event_notify\(\)](#) (p. 2158)) to be notified of the event identified by *flags*.

Returns:

-1 on error; any other value indicates success.

Examples:

```
#include <sys/procmgr.h>

int main ( void )
{
    procmgr_event_trigger( PROCMGR_EVENT_SYNC );
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_guardian()

Let a daemon process take over as a parent

Synopsis:

```
#include <sys/procmgr.h>

pid_t procmgr_guardian( pid_t pid );
```

Arguments:

pid

The ID of the child process that you want to become the guardian of the calling process's other children.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *procmgr_guardian()* allows a daemon process to declare a child process to take over as parent to its children in the event of its death:

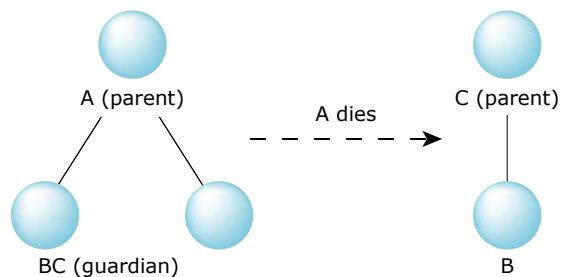


Figure 8: Specifying a guardian for child processes.

Returns:

-1 on error; any other value on success.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
```



```

#include <spawn.h>
#include <sys/procmgr.h>
#include <sys/wait.h>

pid_t    child = -1;
pid_t    guardian = -1;

/*
 * Build a list of the currently running children
 */
void check_children(void) {
    if(child > 0) {
        if(kill(child, 0) == -1) {
            child = -1;
        }
    }
    if(guardian > 0) {
        if(kill(guardian, 0) == -1) {
            guardian = -1;
        }
    }
}

void start_needed_children(void) {
    if(guardian == -1) {
        /* Make a child that will just sit around
         and wait for parent to die */
        while((guardian = fork()) == 0) {
            pid_t parent = getppid();

            /* Wait for parent to die.... */
            fprintf(stderr, "guardian %d waiting on parent %d\n",
                getpid(), parent);

            while(waitpid(parent, 0, 0) != parent);
            /* Then loop around and take over */
        }
        if(guardian == -1) {
            fprintf(stderr, "Unable to start guardian\n");
        } else {
            /* Declare the child a guardian */
            procmgr_guardian(guardian);
        }
    }

    if(child == -1) {
        static char    *args[] = { "sleep", "1000000", 0 };

        if((child = spawnp("sleep", 0, 0, 0, args, 0)) == -1) {
            fprintf(stderr, "Couldn't start child\n");
            child = 0;    /* don't try again */
        }
    }
}

int main(int argc, char *argv[]) {
    fprintf(stderr, "parent %d checking children\n", getpid());
    do {
        fprintf(stderr, "checking children\n");

        /* Learn about the newly adopted children */
        check_children();

        /* If anyone is missing, start them */
        start_needed_children();
    } while(wait(0));    /* Then wait for someone to die... */
    return EXIT_SUCCESS;
}

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_session()

Provide process manager session support

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_session( uint32_t nd,
                    pid_t sid,
                    int id,
                    unsigned event);
```

Arguments:

The interpretation of the arguments depends on the *event*.

nd

A node descriptor. This must currently be ND_LOCAL_NODE, or you'll get an error of ENOREMOTE.

sid

A session ID.

id

A file descriptor, process group, or signal, depending on the event.

event

The event; one of:

- PROCMDR_SESSION_TCSETSID
- PROCMDR_SESSION_SETSID
- PROCMDR_SESSION_SETPGRP
- PROCMDR_SESSION_SIGNAL_PID
- PROCMDR_SESSION_SIGNAL_PGRP
- PROCMDR_SESSION_SIGNAL_LEADER

For more information, see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `procmgr_session()` function provides session support to character device terminal drivers in their resource managers, C library functions, and session management applications.

The arguments that you provide need to match the event; some of these events require certain abilities (see [procmgr_ability\(\)](#) (p. 2141)) be enabled:

PROC_MGR_SESSION_TCSETSID

Used by the [tcsetsid\(\)](#) (p. 3369) function to set the file descriptor, *fd*, to be the controlling terminal for the session headed by the session leader, *sid*.

PROC_MGR_SESSION_SETSID

Used by the [setsid\(\)](#) (p. 2907) function to create a new session with the calling process becoming the session leader. Pass zero for both *sid* and *fd* arguments.

Required ability: `PROC_MGR_AID_PGRP`

PROC_MGR_SESSION_SETPGRP

Used by a character device resource manager to change the process group upon the request of a client calling the [tcsetpgrp\(\)](#) (p. 3367) function. Set the *sid* argument to the client's current session and the *fd* argument to the new target process group for the client.

Required ability: `PROC_MGR_AID_SESSION`

**PROC_MGR_SESSION_SIGNAL_PID, PROC_MGR_SESSION_SIGNAL_PGRP,
PROC_MGR_SESSION_SIGNAL_LEADER**

Used by a character device resource manager to drop a signal of the type specified as the *fd* argument (generally a terminal/job control signal) on the appropriate member of the session specified by the *sid* argument.

Required ability: `PROC_MGR_AID_SESSION`

Returns:

0

Success.

-1

Failure ([errno](#) (p. 539) is set).

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

procmgr_timer_tolerance()

Get or set the default timer tolerance for a process

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_timer_tolerance( const pid_t pid,
                             const uint64_t * const ntime,
                             uint64_t *otime );
```

Arguments:

pid

The process ID of the process whose default timer tolerance you want to set, or 0 to set it for the calling process.

ntime

NULL, or the default timer tolerance, in nanoseconds. Specify a value of ~0ULL for infinite tolerance.

otime

NULL, or a pointer to a location where the function can store the current or previous timer tolerance.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *procmgr_timer_tolerance()* function sets or gets the default timer tolerance for a process. This tolerance is used when calculating timer tolerance if the timer doesn't already have a tolerance set, or has the `TIMER_PRECISE` flag set. This includes the implicit timer used for [TimerTimeout\(\)](#) (p. 3464) operations.

In order to set the default tolerance of another process, the calling process must have the same user ID, be root, or possess the `PROC_MGR_AID_DEFAULT_TIMER_TOLERANCE` ability. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The default timer tolerance is inherited across a [fork\(\)](#) (p. 685), but not an [exec*\(\)](#) (p. 487) or a [spawn*\(\)](#) (p. 2699).

Returns:

0 on success, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EPERM**

Insufficient permission.

ESRCH

There's no process with the specified process ID.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_value_current()

Get the current value of an item being monitored

Synopsis:

```
#include <pthread.h>

uint64_t procmgr_value_current( int id );
```

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Arguments:

id

The ID of a notification added with [procmgr_value_notify_add\(\)](#) (p. 2181).

Description:

The *procmgr_value_current()* function returns the current value of the item being monitored by the notification with the given ID.

Returns:

The value of the monitored item, or ~(uint64_t)0 if an error occurred ([errno](#) (p. 539) is set).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

procmgr_value_notify_add()

Add an event to be triggered when a monitored value crosses a trigger point

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_value_notify_add( unsigned type,
                             int sub_id,
                             uint64_t value,
                             const struct sigevent *event );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Arguments:

type

The type of event you want to add; one of the following:

- PROC_MGR_VALUE_FREE_MEM—the amount of system free memory
- PROC_MGR_VALUE_PRIVATE_MEM—the amount of MAP_PRIVATE memory for a process

You can OR this with one of the following:

- PROC_MGR_VALUE_TRIGGER_UP—deliver the event when the value rises to the trigger point
- PROC_MGR_VALUE_TRIGGER_DOWN—deliver the event when the value drops to the trigger point

sub_id

A sub-ID associated with the *type*:

- For PROC_MGR_VALUE_FREE_MEM, specify 0 for the total memory, or 1 for resident memory.
- For PROC_MGR_VALUE_PRIVATE_MEM, specify the process ID of the process.

value

The trigger point.

event

A pointer to the [sigevent](#) (p. 2961) that you want to be delivered.

Description:

The *procmgr_value_notify_add()* function sets up an event to be triggered when the value being monitored crosses the trigger point up (PROC_MGR_VALUE_TRIGGER_UP) or down (PROC_MGR_VALUE_TRIGGER_DOWN).

For example, PROC_MGR_VALUE_FREE_MEM | PROC_MGR_VALUE_TRIGGER_DOWN causes the event to be delivered when the system free memory drops to the specified trigger value.

If you just want to read the value with *procmgr_value_current()* (p. 2180), don't specify either of the trigger bits.

To delete the notification, call *procmgr_event_notify_delete()* (p. 2168), passing it the ID that *procmgr_value_notify_add()* returns.

Returns:

An event ID, or -1 if there was an error ([errno](#) (p. 539) is set).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

__progrname

The basename of the program being executed

Synopsis:

```
char * __progrname
```

Description:

This global variable holds the basename of the program being executed.



This variable isn't defined in any header file. If you want to refer to it, you need to add your own extern statement.

Classification:

[QNX Neutrino](#) (p. 46)

protoent

Structure for information from the protocol database

Synopsis:

```
#include <netdb.h>

struct protoent {
    char *   p_name;
    char **  p_aliases;
    int      p_proto;
};
```

Description:

The `protoent` structure holds information from the network protocols database, `/etc/protocols`. The members of this structure are:

p_name

The name of the protocol.

p_aliases

A zero-terminated list of alternate names for the protocol.

p_proto

The protocol number.

Classification:

[*POSIX 1003.1*](#) (p. 46)

pthread_abort()

Unconditionally terminate the target thread

Synopsis:

```
#include <pthread.h>

int pthread_abort( pthread_t thread );
```

Arguments:

thread

The ID of the thread that you want to terminate, which you can get when you call [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_abort()` function terminates the target thread. Termination takes effect immediately and isn't a function of the cancelability state of the target thread. No cancellation handlers or thread-specific-data destructor functions are executed. Thread abortion doesn't release any application-visible process resources, including, but not limited to, mutexes and file descriptors. (The behavior of POSIX calls following a call to `pthread_abort()` is unspecified.)

The status of `PTHREAD_ABORTED` is available to any thread joining with the target thread. The constant `PTHREAD_ABORTED` expands to a constant expression, of type `void *`. Its value doesn't match any pointer to an object in memory, or the values `NULL` and `PTHREAD_CANCELED`.

The side effects of aborting a thread that's suspended during a call of a POSIX 1003.1 function are the same as the side effects that may be seen in a single-threaded process when a call to a POSIX 1003.1 function is interrupted by a signal and the given function returns `EINTR`. Any such side effects occur before the thread terminates.

Returns:

EOK

Success.

ESRCH

No thread with the given ID was found.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_atfork()

Register fork handlers

Synopsis:

```
#include <process.h>

int pthread_atfork( void (*prepare)(void),
                   void (*parent)(void),
                   void (*child)(void) );
```

Arguments:

prepare

NULL, or a pointer to the handler to call before the fork.

parent

NULL, or a pointer to the handler to call after the fork in the parent process.

child

NULL, or a pointer to the handler to call after the fork in the child process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_atfork()` function registers fork handler functions to be called before and after a [fork\(\)](#) (p. 685), in the context of the thread that called `fork()`. You can set one or more of the arguments to NULL to indicate no handler.

You can register multiple *prepare*, *parent*, and *child* fork handlers, by making additional calls to `pthread_atfork()`. In this case, the *parent* and *child* handlers are called in the order they were registered, and the *prepare* handlers are called in the reverse order.

Returns:

EOK

Success.

ENOMEM

Insufficient memory to record fork handlers.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_destroy()

Destroy a thread-attribute object

Synopsis:

```
#include <pthread.h>

int pthread_attr_destroy( pthread_attr_t * attr );
```

Arguments:

attr

A pointer to the pthread_attr_t structure that you want to destroy.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_attr_destroy()* function destroys the given thread-attribute object.



The QNX Neutrino implementation of this function doesn't actually free the memory used by the pthread_attr_t structure. To conform to the POSIX standard, don't reuse the attribute object unless you reinitialize it by calling [pthread_attr_init\(\)](#) (p. 2213).

You can use a thread-attribute object to define the attributes of new threads when you call [pthread_create\(\)](#) (p. 2284).

Returns:

0 for success, or an error number.

Errors:

EOK

Success.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getdetachstate()

Get thread detach state attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_getdetachstate(
    const pthread_attr_t* attr,
    int* detachstate );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

detachstate

A pointer to a location where the function can store the thread detach state. For more information, see [pthread_attr_setdetachstate\(\)](#) (p. 2215).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getdetachstate()` function gets the thread detach state attribute from the thread attribute object `attr` and returns it in `detachstate`.

Returns:

EOK

Success.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getguardsize()

Get the size of the thread's guard area

Synopsis:

```
#include <pthread.h>

int pthread_attr_getguardsize(
    const pthread_attr_t* attr,
    size_t* guardsize );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

guardsize

A pointer to a location where the function can store the size of the thread's guard area. For more information, see [*pthread_attr_setguardsize\(\)*](#) (p. 2217).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_attr_getguardsize()* function gets the value of the thread `guardsize` attribute from the attribute structure *attr*.

Returns:

EOK

Success.

EINVAL

Invalid pointer, *attr*, to a `pthread_attr_t` structure.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getinheritsched()

Get a thread's inherit-scheduling attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_getinheritsched(
    const pthread_attr_t* attr,
    int* inheritsched );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

inheritsched

A pointer to a location where the function can store the value of the inherit-scheduling attribute. For more information, see [pthread_attr_setinheritsched\(\)](#) (p. 2219).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getinheritsched()` function gets the thread inherit-scheduling attribute from the attribute object `attr` and returns it in `inheritsched`.

The inherit-scheduling attribute determines whether a thread inherits the scheduling policy of its parent.

Returns:

EOK

Success.

Classification:

POSIX 1003.1 THR TPS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getschedparam()

Get thread scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedparam(
    const pthread_attr_t * attr,
    struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

param

A pointer to a [sched_param](#) (p. 2731) structure where the function can store the current scheduling parameters.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getschedparam()` function gets the thread scheduling parameters attribute from the thread attribute object `attr` and returns it in `param`.

Returns:

EOK

Success.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getschedpolicy()

Get thread scheduling policy attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedpolicy(
    const pthread_attr_t* attr,
    int* policy );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

policy

The current thread scheduling policy. For more information, see [*pthread_attr_setschedpolicy\(\)*](#) (p. 2223).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_attr_getschedpolicy()* function gets the thread scheduling policy attribute from the thread attribute object *attr* and returns it in *policy*.

Returns:

EOK

Success.

Classification:

[*POSIX 1003.1 THR TPS*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getscope()

Get thread contention scope attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_getscope(
    const pthread_attr_t *attr,
    int *scope );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

scope

A pointer to a location where the function can store the current contention scope. For more information, see [pthread_attr_setscope\(\)](#) (p. 2225).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getscope()` function gets the thread contention scope attribute from the thread attribute object `attr` and returns it in `scope`.

Returns:

EOK

Success.

Classification:

[POSIX 1003.1 THR TPS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getstack()

Get the thread-creation stack attributes

Synopsis:

```
#include <pthread.h>

int pthread_attr_getstack( const pthread_attr_t *attr,
                          void **stackaddr,
                          size_t *stacksize );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

stackaddr

A pointer to a location where the function can store the base address of the thread stack.

stacksize

A pointer to a location where the function can store the size of the thread stack, in bytes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getstack()` function gets the stack attributes from the given thread attributes.

Returns:

`EOK`

Classification:

POSIX 1003.1 TSA TSS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getstackaddr()

Get thread stack address attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_getstackaddr(
    const pthread_attr_t* attr,
    void** stackaddr );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

stackaddr

A pointer to a location where the function can store the address of the thread stack.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getstackaddr()` function gets the thread stack address attribute from the thread attribute object `attr` and returns it in `stackaddr`.

For more information about the thread stack, see [pthread_attr_setstackaddr\(\)](#) (p. 2229)

Returns:

EOK

Success.

Classification:

[POSIX 1003.1 THR TSA](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getstacklazy()

Get the thread lazy-stack attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_getstacklazy(
    const pthread_attr_t * attr,
    int *lazystack );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

lazystack

A pointer to a location where the function can store the current lazy-stack attribute. For more information, see [pthread_attr_setstacklazy\(\)](#) (p. 2231).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getstacklazy()` function gets the thread lazy-stack attribute in the attribute object `attr` and stores it in the location pointed to by `lazystack`.

Returns:

EOK

Success.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getstackprealloc()

Get the amount of memory to preallocate for a thread's MAP_LAZY stack

Synopsis:

```
#include <pthread.h>

int pthread_attr_getstackprealloc(
    const pthread_attr_t * attr,
    size_t *stacksize);
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

stacksize

A pointer to a location where the function can store the size of the stack.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_attr_getstackprealloc()* function gets the amount of memory to preallocate for a thread's MAP_LAZY stack.



This function was added in the QNX Neutrino Core OS 6.3.2.

Returns:

EOK

Success.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_getstacksize()

Get the thread stack-size attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_getstacksize(
    const pthread_attr_t* attr,
    size_t* stacksize );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

stacksize

A pointer to a location where the function can store the stack size to be used for new threads.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getstacksize()` function gets the thread stack size attribute from the thread attribute object `attr` and returns it in `stacksize`.

You can set the stack size by calling [pthread_attr_setstacksize\(\)](#) (p. 2235).

Returns:

EOK

Success.

Classification:

[POSIX 1003.1 THR TSS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_init()

Initialize a thread-attribute object

Synopsis:

```
#include <pthread.h>

int pthread_attr_init( pthread_attr_t *attr );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that you want to initialize. For more information, see below.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_init()` function initializes the thread attributes in the thread attribute object `attr` to their default values:

- Cancellation requests may be acted on according to the cancellation type (`PTHREAD_CANCEL_DEFERRED`).
- Cancellation requests are held pending until a cancellation point (`PTHREAD_CANCEL_ENABLE`).
- Threads are put into a zombie state when they terminate, and they stay in this state until you retrieve their exit status or detach them (`PTHREAD_CREATE_JOINABLE`).
- Threads inherit the scheduling policy of their parent thread (`PTHREAD_INHERIT_SCHED`).
- Threads are scheduled against all threads in the system (`PTHREAD_SCOPE_SYSTEM`).
- The stack attributes are set so that the kernel will allocate a 4 KB stack for new threads and free the stacks when the threads terminate.

After initialization, you can use the `pthread_attr_*` family of functions to get and set the attributes:

Get	Set
pthread_attr_getdetachstate() (p. 2191)	pthread_attr_setdetachstate() (p. 2215)
pthread_attr_getguardsize() (p. 2193)	pthread_attr_setguardsize() (p. 2217)
pthread_attr_getinheritsched() (p. 2195)	pthread_attr_setinheritsched() (p. 2219)
pthread_attr_getschedparam() (p. 2197)	pthread_attr_setschedparam() (p. 2221)
pthread_attr_getschedpolicy() (p. 2199)	pthread_attr_setschedpolicy() (p. 2223)
pthread_attr_getscope() (p. 2201)	pthread_attr_setscope() (p. 2225)
pthread_attr_getstack() (p. 2203)	pthread_attr_setstack() (p. 2227)
pthread_attr_getstackaddr() (p. 2205)	pthread_attr_setstackaddr() (p. 2229)
pthread_attr_getstacklazy() (p. 2207)	pthread_attr_setstacklazy() (p. 2231)
pthread_attr_getstackprealloc() (p. 2209)	pthread_attr_setstackprealloc() (p. 2233)
pthread_attr_getstacksize() (p. 2211)	pthread_attr_setstacksize() (p. 2235)

You can also set some non-POSIX attributes; for more information, see “[QNX Neutrino extensions](#) (p. 2285),” in the documentation for [pthread_create\(\)](#).

You can then pass the attribute object to [pthread_create\(\)](#) (p. 2284) to create a thread with the required attributes. You can use the same attribute object in multiple calls to [pthread_create\(\)](#).

The effect of initializing an already-initialized thread-attribute object is undefined.

Returns:

EOK

Success.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setdetachstate()

Set thread detach state attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_setdetachstate(
    pthread_attr_t* attr,
    int detachstate );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

detachstate

The new value for the thread detach state:

- `PTHREAD_CREATE_JOINABLE` — create the thread in a joinable state.
- `PTHREAD_CREATE_DETACHED` — create the thread in a detached state.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setdetachstate()` function sets the thread detach state attribute in the thread attribute object `attr` to `detachstate`.

The default value for the thread detach state is `PTHREAD_CREATE_JOINABLE`.

Returns:

EOK

Success.

EINVAL

Invalid thread detach state value.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setguardsize()

Set the size of the thread's guard area

Synopsis:

```
#include <pthread.h>

int pthread_attr_setguardsize(
    pthread_attr_t* attr,
    size_t guardsize );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

guardsize

The new value for the size of the thread's guard area.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setguardsize()` function sets the size of the thread's guard area in the attribute structure `attr` to `guardsize`.

If `guardsize` is 0, threads created with `attr` have no guard area; otherwise, a guard area of at least `guardsize` bytes is provided. You can get the default `guardsize` value by specifying `_SC_PAGESIZE` in a call to [sysconf\(\)](#) (p. 3308).

The `guardsize` attribute controls the size of the guard area for the thread's stack. This guard area helps protect against stack overflows; `guardsize` bytes of extra memory is allocated at the overflow end of the stack. If a thread overflows into this buffer, it receives a SIGSEGV signal.

The `guardsize` attribute is provided because:

- Stack overflow protection can waste system resources. An application that creates many threads can save system resources by turning off guard areas if it trusts its threads not to overflow the stack.
- When threads allocate large objects on the stack, a large *guardsize* is required to detect stack overflows.

Returns:**EOK**

Success.

EINVAL

Invalid pointer, *attr*, to a `pthread_attr_t` structure, or *guardsize* is invalid.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

If you provide a stack (using *attr*'s *stackaddr* attribute; see [pthread_attr_setstackaddr\(\)](#) (p. 2229)), the *guardsize* is ignored, and there's no stack overflow protection for that thread.

The *guardsize* argument is completely ignored when using a physical mode memory manager.

pthread_attr_setinheritsched()

Set a thread's inherit-scheduling attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_setinheritsched(
    pthread_attr_t * attr,
    int inheritsched );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

inheritsched

The new value for the thread's inherit-scheduling attribute:

- `PTHREAD_INHERIT_SCHED` — the thread inherits the scheduling policy of the parent thread.
- `PTHREAD_EXPLICIT_SCHED` — use the scheduling policy specified in *attr* for the thread.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setinheritsched()` function sets the thread inherit scheduling attribute in the attribute object *attr* to *inheritsched*.

The default value of the thread inherit scheduling attribute is `PTHREAD_INHERIT_SCHED`.

Returns:

EOK

Success.

EINVAL

Invalid thread attribute object *attr*.

ENOTSUP

Invalid thread inherit scheduling attribute *inheritsched*.

Classification:

[*POSIX 1003.1 THR TPS*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setschedparam()

Set a thread's scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(
    pthread_attr_t * attr,
    const struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

param

A pointer to a [sched_param](#) (p. 2731) structure that defines the thread's scheduling parameters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setschedparam()` function sets the thread scheduling parameters attribute in the thread attribute object *attr* to *param*.

The thread scheduling parameters are used only if you've set the thread inherit scheduling attribute to `PTHREAD_EXPLICIT_SCHED` by calling [pthread_attr_setinheritsched\(\)](#) (p. 2219). By default, a thread inherits its parent's priority.

Returns:

EOK

Success.

EINVAL

Invalid thread attribute object *attr*.

ENOTSUP

Invalid thread scheduling parameters attribute *param*.

Examples:

See the entry for [sched_param](#) (p. 2733).

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setschedpolicy()

Set the thread scheduling policy attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedpolicy(
    pthread_attr_t* attr,
    int policy );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

policy

The new value for the scheduling policy:

- `SCHED_FIFO` — first-in first-out scheduling.
- `SCHED_RR` — round-robin scheduling.
- `SCHED_OTHER` — currently the same as `SCHED_RR`.
- `SCHED_NOCHANGE` — don't change the policy.
- `SCHED_SPORADIC` — sporadic scheduling.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_attr_setschedpolicy()* function sets the thread scheduling policy attribute in the thread attribute object *attr* to *policy*.

The *policy* attribute is used only if you've set the thread inherit-scheduling attribute to `PTHREAD_EXPLICIT_SCHED` by calling [*pthread_attr_setinheritsched\(\)*](#) (p. 2219).

For descriptions of the scheduling policies, see “Scheduling policies” in the chapter on the microkernel in the *System Architecture* guide.

Returns:**EOK**

Success.

EINVAL

Invalid thread attribute object *attr*.

ENOTSUP

Invalid thread scheduling policy *policy*.

Classification:

[*POSIX 1003.1 THR TPS*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setscope()

Set thread contention scope attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_setscope( pthread_attr_t* attr,
                           int scope );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

scope

The new value for the contention scope attribute:

- `PTHREAD_SCOPE_SYSTEM` — schedule all threads together.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setscope()` sets the thread contention scope attribute in the thread attribute object `attr` to `scope`.

Returns:

EOK

Success.

EINVAL

Invalid thread attribute object `attr`.

ENOTSUP

Invalid thread contention scope attribute *scope*.

Classification:

POSIX 1003.1 THR TPS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setstack()

Set the thread-creation stack attributes

Synopsis:

```
#include <pthread.h>

int pthread_attr_setstack( pthread_attr_t *attr,
                          void *stackaddr,
                          size_t stacksize );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

stackaddr

The base address (i.e. the lowest addressable byte) of the created thread's stack.

stacksize

The size of the thread stack, in bytes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setstack()` function sets the stack attributes that specify the area of storage to be used for the created thread's stack. The value of `stacksize` must be at least `PTHREAD_STACK_MIN`.

Returns:

EOK

Success.

EINVAL

The *stacksize* is less than PTHREAD_STACK_MIN.

Classification:

POSIX 1003.1 TSA TSS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setstackaddr()

Set the thread stack address attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_setstackaddr( pthread_attr_t * attr,
                               void * stackaddr );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

stackaddr

A pointer to the block of memory that you want a new thread to use as its stack.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setstackaddr()` function sets the thread stack address attribute in the attribute object `attr` to `stackaddr`.

The default value for the thread stack address attribute is `NULL`. A thread created with a `NULL` stack address attribute will have a stack dynamically allocated by the system of minimum size `PTHREAD_STACK_MIN`. If the system allocates a stack, it reclaims the space when the thread terminates. If you allocate a stack, you must free it.



- If you provide a stack, the `guardsize` member of the `pthread_attr_t` structure is ignored, and there's no stack overflow protection for the thread.
 - The system uses some of the provided stack for objects such as thread local storage and an initial stack frame, so less than the entire buffer is available to the thread.
-

Returns:**EOK**

Success.

Classification:*POSIX 1003.1 THR TSA* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The QNX Neutrino interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```
void nothingthread( void )
{
    return;
}
```

pthread_attr_setstacklazy()

Set the thread lazy-stack attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_setstacklazy(
    pthread_attr_t * attr,
    int lazystack );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

lazystack

One of:

- `PTHREAD_STACK_LAZY` — allocate physical memory for the thread stack on demand (the default).
- `PTHREAD_STACK_NOTLAZY` — allocate physical memory for the whole stack up front. Use this value to ensure that your server processes don't die later on because they're unable to allocate stack memory. We recommend that you set the stack size as well, because the default stack size is probably much larger than you really need.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setstacklazy()` function sets the thread stack attribute in the attribute object `attr` to `lazystack`.

Returns:

EOK

Success.

EINVAL

The thread-attribute object that *attr* points to is invalid.

ENOTSUP

The value of *lazystack* is invalid.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setstackprealloc()

Set the amount of memory to preallocate for a thread's MAP_LAZY stack

Synopsis:

```
#include <pthread.h>

int pthread_attr_setstackprealloc(
    const pthread_attr_t * attr,
    size_t stacksize);
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [*pthread_attr_init\(\)*](#) (p. 2213).

stacksize

The amount of stack you want to preallocate for new threads.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_attr_setstackprealloc()* function sets the size of the memory to preallocate for a thread's MAP_LAZY stack.

By default, the system allocates `sysconf(_SC_PAGESIZE)` bytes of physical memory for the initial stack reference. This function allows you to change this default memory size, if you know that a thread will need more stack space. Semantically, there is no difference in operation, but the memory manager attempts to make more efficient use of Memory Management Unit hardware (e.g. a larger page size in the page entry table) for the stack if it knows upfront that more memory will be required.



This function was added in the QNX Neutrino Core OS 6.3.2.

Returns:**EOK**

Success.

Classification:[*QNX Neutrino*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_attr_setstacksize()

Set the thread stack-size attribute

Synopsis:

```
#include <pthread.h>

int pthread_attr_setstacksize( pthread_attr_t * attr,
                               size_t stacksize );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see [pthread_attr_init\(\)](#) (p. 2213).

stacksize

The size of the stack you want to use in new threads. The minimum value of the thread stack-size attribute is `PTHREAD_STACK_MIN`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setstacksize()` function sets the thread stack size attribute in the thread attribute object `attr` to `stacksize`.



The system uses some of the provided stack for objects such as thread local storage and an initial stack frame, so less than the entire buffer is available to the thread.

Returns:

EOK

Success.

EINVAL

The value of *stacksize* is less than PTHREAD_STACK_MIN or greater than the system limit.

Classification:

POSIX 1003.1 THR TSS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The QNX Neutrino interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```
void nothingthread( void )
{
    return;
}
```

pthread_barrier_destroy()

Destroy a barrier object

Synopsis:

```
#include <pthread.h>

int pthread_barrier_destroy(
    pthread_barrier_t * barrier );
```

Arguments:

barrier

A pointer to the pthread_barrier_t object that you want to destroy.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_barrier_destroy()* function destroys the barrier referenced by *barrier* and releases any resources used by the barrier. Subsequent use of the barrier is undefined until you reinitialize the barrier by calling [pthread_barrier_init\(\)](#) (p. 2239).

Returns:

EBUSY

The *barrier* is in use.

EINVAL

Invalid *barrier*.

EOK

Success.

Classification:

[POSIX 1003.1 THR BAR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_barrier_init()

Initialize a barrier object

Synopsis:

```
#include <pthread.h>

int pthread_barrier_init(
    pthread_barrier_t * barrier,
    const pthread_barrierattr_t * attr
    unsigned int count );
```

Arguments:

barrier

A pointer to the pthread_barrier_t object that you want to initialize.

attr

NULL, or a pointer to a pthread_barrierattr_t structure that specifies the attributes that you want to use for the barrier.

count

The number of threads that must call [pthread_barrier_wait\(\)](#) (p. 2241) before any of them successfully returns from the call. This value must be greater than zero.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_barrier_init()* function allocates any resources required to use the barrier referenced by *barrier* and initializes the barrier with attributes referenced by *attr*. If *attr* is NULL, the default barrier attributes are used. The effect is the same as passing the address of a default barrier attributes object. Once it's initialized, you can use the barrier any number of times without reinitializing it.



You should allocate synchronization objects only in normal memory mappings. On certain processors, the atomic operations performed on synchronization

objects will cause a fault if the control structure is allocated in uncached memory.

If *pthread_barrier_init()* fails, the barrier isn't initialized.

In cases where the default barrier attributes are appropriate, you can use *PTHREAD_BARRIER_INITIALIZER()* macro to initialize barriers that are statically allocated. The effect is equivalent to dynamic initialization by a call to *pthread_barrier_init()* with parameter *attr* specified as *NULL*, except that no error checks are performed.

Returns:

EAGAIN

The system lacks the necessary resources to initialize another barrier.

EBUSY

Attempt to reinitialize a barrier while it's in use.

EFAULT

A fault occurred when the kernel tried to access *barrier* or *attr*.

EINVAL

Invalid value specified by *attr*.

EOK

Success.

Classification:

[*POSIX 1003.1 THR BAR*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_barrier_wait()

Synchronize participating threads at the barrier

Synopsis:

```
#include <sync.h>

int pthread_barrier_wait( pthread_barrier_t * barrier );
```

Arguments:

barrier

A pointer to the `pthread_barrier_t` object that you want to use to synchronize the threads. You must initialize the barrier by calling [*pthread_barrier_init\(\)*](#) (p. 2239) before calling *pthread_barrier_wait()*.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_barrier_wait()* function synchronizes participating threads at the barrier referenced by *barrier*. The calling thread blocks — that is, it doesn't return from *pthread_barrier_wait()* — until the required number of threads have called *pthread_barrier_wait()*, specifying the barrier.

When the required number of threads have called *pthread_barrier_wait()* specifying the barrier, the function returns `PTHREAD_BARRIER_SERIAL_THREAD` to one unspecified thread, and zero to each of the remaining threads. At this point, the barrier is reset to the state it occupied as a result of the most recent *pthread_barrier_init()* function that referenced it.

The constant `PTHREAD_BARRIER_SERIAL_THREAD` is defined in `<pthread.h>`, and its value is distinct from any other value that *pthread_barrier_wait()* returns.

If a signal is delivered to a thread blocked on a barrier, on return from the signal handler, the thread resumes waiting at the barrier as if it hadn't been interrupted.

Returns:

PTHREAD_BARRIER_SERIAL_THREAD to a single (arbitrary) thread synchronized at the barrier, and zero to each of the other threads; otherwise, *pthread_barrier_wait()* returns an error number:

EINVAL

The *barrier* argument isn't initialized.

Classification:

[*POSIX 1003.1 THR BAR*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_barrierattr_destroy()

Destroy a barrier-attributes object

Synopsis:

```
#include <pthread.h>

int pthread_barrierattr_destroy(
    pthread_barrierattr_t * attr );
```

Arguments:

attr

A pointer to the `pthread_barrierattr_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_barrierattr_destroy()` function destroys the barrier-attributes object *attr*. Subsequent use of the object is undefined until you reinitialize the object by calling [pthread_barrierattr_init\(\)](#) (p. 2247).

Once you've used a barrier-attributes object to initialize one or more barriers, any changes to the attributes object (including destroying it) don't affect any previously initialized barriers.

Returns:

EOK

Success.

EINVAL

Invalid barrier attribute object *attr*.

Classification:

[POSIX 1003.1 THR BAR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_barrierattr_getpshared()

Get the value of a barrier's process-shared attribute

Synopsis:

```
#include <pthread.h>

int pthread_barrierattr_getpshared(
    const pthread_barrierattr_t * attr
    int * pshared );
```

Arguments:

attr

A pointer to the `pthread_barrierattr_t` object whose attribute you want to query. You must have initialized this object by calling [*pthread_barrierattr_init\(\)*](#) (p. 2247) before calling *pthread_barrierattr_getpshared()*.

pshared

A pointer to a location where the function can store the value of the process-shared attribute. For information about the possible values, see [*pthread_barrierattr_setpshared\(\)*](#) (p. 2249).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_barrierattr_getpshared()* function gets the value of the process-shared attribute from the attributes object referenced by *attr* and stores the value in the object referenced by *pshared*.

Returns:

EOK

Success.

EINVAL

Invalid barrier attribute object *attr*.

Classification:

POSIX 1003.1 THR BAR TSH (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_barrierattr_init()

Initialize a barrier attributes object

Synopsis:

```
#include <pthread.h>

int pthread_barrierattr_init(
    pthread_barrierattr_t * attr );
```

Arguments:

attr

A pointer to the `pthread_barrierattr_t` object that you want to initialize.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_barrierattr_init()` function initializes the barrier attributes object `attr` with the default value for all of the attributes.

Returns:

EOK

Success.

ENOMEM

Insufficient memory to initialize barrier attribute object `attr`.

Classification:

[*POSIX 1003.1 THR BAR*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

pthread_barrierattr_setpshared()

Set the value of a barrier's process-shared attribute

Synopsis:

```
#include <sync.h>

int pthread_barrierattr_setpshared(
    pthread_barrierattr_t * attr
    int pshared );
```

Arguments:

attr

A pointer to the `pthread_barrierattr_t` object whose attribute you want to set. You must have initialized this object by calling [*pthread_barrierattr_init\(\)*](#) (p. 2247) before calling *pthread_barrierattr_setpshared()*.

pshared

The new value of the process-shared attribute; one of:

- `PTHREAD_PROCESS_SHARED` — allow a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated.
- `PTHREAD_PROCESS_PRIVATE` (default behavior) — allow a barrier to be operated on only by threads created within the same process as the thread that initialized the barrier. If threads of different processes attempt to operate on such a barrier, the behavior is as if `PTHREAD_PROCESS_SHARED` were set.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_barrierattr_setpshared()* function sets the process-shared attribute in an initialized attributes object referenced by *attr*.

Returns:

EOK

Success.

EINVAL

The attribute object, *attr*, or the new value specified in *pshared* isn't valid.

Classification:

[*POSIX 1003.1 THR BAR TSH*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cancel()

Cancel a thread

Synopsis:

```
#include <pthread.h>

int pthread_cancel( pthread_t thread );
```

Arguments:

thread

The ID of the thread that you want to cancel, which you can get when you call [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cancel()` function requests that the target thread *thread* be canceled (terminated). The cancellation type and state of the target determine when the cancellation takes effect.

When the cancellation is acted on, the target's cancellation cleanup handlers are called. When the last cancellation cleanup handler returns, the target's thread-specific-data destructor functions are called. When the last destructor function returns, the target is terminated. Cancellation processing in the target thread runs asynchronously with respect to the calling thread.

Returns:

EOK

Success.

ESRCH

No thread with thread ID *thread* exists.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cleanup_pop()

Pop a function off of a thread's cancellation-cleanup stack

Synopsis:

```
#include <pthread.h>

void pthread_cleanup_pop( int execute );
```

Arguments:

execute

Zero if you don't want to execute the handler; nonzero if you do.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cleanup_pop()` macro pops the top cancellation-cleanup handler from the calling thread's cancellation-cleanup stack and invokes the handler if *execute* is nonzero.



The `pthread_cleanup_pop()` macro expands to a few lines of code that end with a closing brace `}`, but don't have a matching opening brace `{`. You must pair `pthread_cleanup_pop()` with [pthread_cleanup_push\(\)](#) (p. 2255) within the same lexical scope.

Examples:

See [pthread_cleanup_push\(\)](#) (p. 2255).

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

pthread_cleanup_push()

Push a function onto a thread's cancellation-cleanup stack

Synopsis:

```
#include <pthread.h>

void pthread_cleanup_push( void (routine)(void*),
                          void* arg );
```

Arguments:

routine

The handler that you want to push onto the thread's stack.

arg

A pointer to whatever data you want to pass to the function when it's called.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cleanup_push()` function pushes the given cancellation-cleanup handler *routine* onto the calling thread's cancellation-cleanup stack.

The cancellation-cleanup handler is popped from the stack and invoked with argument *arg* when the thread:

- exits (i.e. calls [pthread_exit\(\)](#) (p. 2292))
- acts on a cancellation request
- calls [pthread_cleanup_pop\(\)](#) (p. 2253) with a nonzero argument.



The `pthread_cleanup_push()` macro expands to a few lines of code that start with an opening brace (`{`), but don't have a matching closing brace (`}`). You must pair `pthread_cleanup_push()` with `pthread_cleanup_pop()` within the same lexical scope.

Examples:

Use a cancellation cleanup handler to free resources, such as a mutex, when a thread is terminated:

```
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void unlock( void * arg )
{
    pthread_mutex_unlock( &lock );
}

void * function( void * arg )
{
    while( 1 )
    {
        pthread_mutex_lock( &lock );
        pthread_cleanup_push( &unlock, NULL );

        /*
         * Any of the possible cancellation points could
         * go here.
         */
        pthread_testcancel();

        pthread_cleanup_pop( 1 );
    }
}
```

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cond_broadcast()

Unblock threads waiting on condition

Synopsis:

```
#include <pthread.h>

int pthread_cond_broadcast( pthread_cond_t* cond );
```

Arguments:

cond

A pointer to the `pthread_cond_t` object for which you want to unblock the threads.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cond_broadcast()` function unblocks all threads currently blocked on the condition variable `cond`. The threads are unblocked in priority order.

If more than one thread at a particular priority is blocked, those threads are unblocked in FIFO order.

Returns:

EOK

Success.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

Invalid condition variable `cond`.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cond_destroy()

Destroy condition variable

Synopsis:

```
#include <pthread.h>

int pthread_cond_destroy( pthread_cond_t* cond );
```

Arguments:

cond

A pointer to the `pthread_cond_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cond_destroy()` function destroys the condition variable *cond*. After you've destroyed a condition variable, you shouldn't reuse it until you've reinitialized it by calling [pthread_cond_init\(\)](#) (p. 2261).

Any other threads that are blocked on the condition variable become unblocked.

Returns:

EOK

Success.

EINVAL

Invalid condition variable *cond*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cond_init()

Initialize a condition variable

Synopsis:

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init( pthread_cond_t *cond,
                      const pthread_condattr_t *attr );
```

Arguments:

cond

A pointer to the pthread_cond_t object that you want to initialize.

attr

NULL, or a pointer to a pthread_condattr_t object that specifies the attributes that you want to use for the condvar. For more information, see [pthread_condattr_init\(\)](#) (p. 2278).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_cond_init()* function initializes the condition variable *cond* with the attributes in the condition variable attribute object *attr*. If *attr* is NULL, *cond* is initialized with the default values for the attributes.



You should allocate synchronization objects only in normal memory mappings. On certain processors, atomic operations such as calls to [pthread_mutex_lock\(\)](#) (p. 2318) will cause a fault if the control structure is allocated in uncached memory.

If the condition variable is statically allocated, you can initialize it with the default attribute values by assigning to it the macro PTHREAD_COND_INITIALIZER.

Condition variables have at least the following attributes defined:

PTHREAD_PROCESS_PRIVATE

The condition variable can only be accessed by threads created within the same process as the thread that initialized the condition variable.

PTHREAD_PROCESS_SHARED

Any thread that has access to the memory where the condition variable is allocated can access the condition variable.

For more information about these attributes, see [pthread_condattr_getpshared\(\)](#) (p. 2276) and [pthread_condattr_setpshared\(\)](#) (p. 2282).

Returns:**EOK**

Success.

EAGAIN

All kernel synchronization objects are in use.

EBUSY

The *cond* argument pointer to a previously initialized condition variable that hasn't been destroyed.

EFAULT

A fault occurred when the kernel tried to access *cond* or *attr*.

EINVAL

The value specified by *cond* is invalid.

ENOMEM

There wasn't enough memory to initialize the condvar.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cond_signal()

Unblock a thread that's waiting on a condition variable

Synopsis:

```
#include <pthread.h>

int pthread_cond_signal( pthread_cond_t* cond );
```

Arguments:

cond

A pointer to the `pthread_cond_t` object for which you want to unblock the highest-priority thread.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_cond_signal()* function unblocks the highest priority thread that's waiting on the condition variable, *cond*.

If more than one thread at the highest priority is waiting, *pthread_cond_signal()* unblocks the one that has been waiting the longest.

Returns:

EOK

Success.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

Invalid condition variable *cond*.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cond_timedwait()

Wait on a condition variable, with a time limit

Synopsis:

```
#include <pthread.h>
#include <time.h>

int pthread_cond_timedwait(
    pthread_cond_t* cond,
    pthread_mutex_t* mutex,
    const struct timespec* abstime );
```

Arguments:

cond

The condition variable on which to block the thread.

mutex

The mutex associated with the condition variable.

abstime

A pointer to a [timespec](#) (p. 3473) structure that specifies the maximum time to block the thread, expressed as an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cond_timedwait()` function blocks the calling thread on the condition variable *cond*, and unlocks the associated mutex *mutex*. The calling thread must have locked *mutex* before waiting on the condition variable. Upon return from the function, the mutex is again locked and owned by the calling thread.

The calling thread is blocked until either another thread performs a signal or broadcast on the condition variable, the absolute time specified by *abstime* has passed, a signal is delivered to the thread, or the thread is canceled (waiting on a condition variable is a cancellation point). In all cases, the thread reacquires the mutex before being unblocked.

To specify the clock source for the condvar, call [pthread_condattr_setclock\(\)](#) (p. 2280). As discussed in “Advanced topics” in the Clocks, Timers, and Getting a Kick Every So Often chapter of *Get Programming with the QNX Neutrino*, `CLOCK_MONOTONIC` isn't affected by changes to the system time; the default, `CLOCK_REALTIME`, is.



Don't use a recursive mutex with condition variables.

If a thread that's blocked on a condition variable is canceled, the thread reacquires the mutex that's guarding the condition variable, so that the thread's cleanup handlers run in the same state as the critical code before and after the call to this function. If some other thread owns the lock, the canceled thread blocks until the mutex is available.



- Make sure that the thread's cleanup handlers unlock the mutex.
 - To specify a timeout of less than a second, obtain the current time using [clock_gettime\(\)](#) (p. 312), and then add the number of nanoseconds to the `abstime.tv_nsec` field.
 - You can use [nsec2timespec\(\)](#) (p. 1895) to convert times in nanoseconds to a `timespec` structure (and [timespec2nsec\(\)](#) (p. 3474) to convert them back again).
-

Returns:

EOK

Success, or the call was interrupted by a signal.

EAGAIN

Insufficient system resources are available to wait on the condition.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

One or more of the following is true:

- One or more of `cond`, `mutex`, and `abstime` is invalid.
- Concurrent waits or timed waits on `cond` used different mutexes.
- The mutex has died; see [SyncMutexEvent\(\)](#) (p. 3291).

EPERM

The current thread doesn't own the mutex.

ETIMEDOUT

The time specified by *abstime* has passed.

Examples:

Wait five seconds while trying to acquire control over a condition variable:

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c;

int main(int argc, char* argv[])
{
    struct timespec to;
    int retval;
    pthread_condattr_t attr;

    fprintf(stderr, "starting...\n");

    /* Set up the condvar attributes to use CLOCK_MONOTONIC. */
    pthread_condattr_init( &attr);
    pthread_condattr_setclock( &attr, CLOCK_MONOTONIC);
    pthread_cond_init( &c, &attr);

    /*
     Here's the interesting bit; we'll wait for
     five seconds FROM NOW when we call
     pthread_cond_timedwait().
    */
    clock_gettime(CLOCK_MONOTONIC, &to);
    to.tv_sec += 5;

    if ((retval = pthread_mutex_lock(&m))) {
        fprintf(stderr, "pthread_mutex_lock %s\n",
            strerror(retval));

        exit(EXIT_FAILURE);
    }

    if ((retval = pthread_cond_timedwait(&c, &m, &to)))
    {
        fprintf(stderr, "pthread_cond_timedwait %s\n",
            strerror(retval));

        exit(EXIT_FAILURE);
    }

    if ((retval = pthread_mutex_unlock(&m))) {
        fprintf(stderr, "pthread_mutex_unlock %s\n",
            strerror(retval));

        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_cond_wait()

Wait on condition variable

Synopsis:

```
#include <pthread.h>

int pthread_cond_wait( pthread_cond_t* cond,
                      pthread_mutex_t* mutex );
```

Arguments:

cond

A pointer to the `pthread_cond_t` object that you want the threads to block on.

mutex

The mutex that you want to unlock.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_cond_wait()` function blocks the calling thread on the condition variable `cond`, and unlocks the associated mutex `mutex`. The calling thread must have locked `mutex` before waiting on the condition variable. On return from the function, the mutex is again locked and owned by the calling thread.

The calling thread is blocked until either another thread performs a signal or broadcast on the condition variable, a signal is delivered to the thread, or the thread is canceled (waiting on a condition variable is a cancellation point). In all cases, the thread reacquires the mutex before being unblocked.

If a thread that's blocked on a condition variable is canceled, the thread reacquires the mutex that's guarding the condition variable, so that the thread's cleanup handlers run in the same state as the critical code before and after the call to this function. If some other thread owns the lock, the canceled thread blocks until the mutex is available.



- Don't use a recursive mutex with condition variables.
- Make sure that the thread's cleanup handlers unlock the mutex.
- This function can interfere with the kernel's efforts to manage power usage. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Returns:

EOK

Success, or the call was interrupted by a signal.

EAGAIN

Insufficient system resources are available to wait on the condition.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

One or more of the following is true:

- One or more of *cond* or *mutex* is invalid.
- Concurrent waits on *cond* used different mutexes.
- The mutex has died; see [SyncMutexEvent\(\)](#) (p. 3291).

EPERM

The current thread doesn't own the mutex.

Examples:

Use condition variables to synchronize producer and consumer threads:

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int condition = 0;
int count = 0;

int consume( void )
{
    while( 1 )
    {
        pthread_mutex_lock( &mutex );
        while( condition == 0 )
            pthread_cond_wait( &cond, &mutex );
        printf( "Consumed %d\n", count );
        condition = 0;
        pthread_cond_signal( &cond );
        pthread_mutex_unlock( &mutex );
    }
}
```

```

    return( 0 );
}

void*  produce( void * arg )
{
    while( 1 )
    {
        pthread_mutex_lock( &mutex );
        while( condition == 1 )
            pthread_cond_wait( &cond, &mutex );
        printf( "Produced %d\n", count++ );
        condition = 1;
        pthread_cond_signal( &cond );
        pthread_mutex_unlock( &mutex );
    }
    return( 0 );
}

int main( void )
{
    pthread_create( NULL, NULL, &produce, NULL );
    return consume();
}

```

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_condattr_destroy()

Destroy a condition-variable attribute object

Synopsis:

```
#include <pthread.h>

int pthread_condattr_destroy(
    pthread_condattr_t* attr );
```

Arguments:

attr

A pointer to the pthread_condattr_t object that you want to destroy.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The pthread_condattr_destroy() function destroys the condition variable attribute object attr. Once you've destroyed the condition-variable attribute object, don't reuse it until you've reinitialized it by calling pthread_condattr_init() (p. 2278).

Returns:

EOK

Success.

EINVAL

Invalid condition variable attribute object attr.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

pthread_condattr_getclock()

Get the clock attribute from a condition-variable attribute object

Synopsis:

```
#include <pthread.h>

int pthread_condattr_getclock(
    const pthread_condattr_t * attr,
    clockid_t * id );
```

Arguments:

attr

A pointer to the `pthread_condattr_t` object from which you want to get the clock.

id

A pointer to a `clockid_t` object where the function can store the clock ID.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_condattr_getclock()` function obtains the value of the clock attribute from the attributes object referenced by `attr`.

The clock attribute is the clock ID of the clock that's used to measure the timeout service of [pthread_cond_timedwait\(\)](#) (p. 2265); one of:

- `CLOCK_REALTIME` — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- `CLOCK_SOFTTIME` — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a `CLOCK_SOFTTIME` timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, `CLOCK_SOFTTIME` behaves the same as `CLOCK_REALTIME`.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

The default value of the clock attribute refers to the system clock.

Returns:**EOK**

Success.

EINVAL

Invalid value *attr*.

Classification:

[POSIX 1003.1 THR CS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_condattr_getpshared()

Get the process-shared attribute from a condition variable attribute object

Synopsis:

```
#include <pthread.h>

int pthread_condattr_getpshared(
    const pthread_condattr_t* attr,
    int* pshared );
```

Arguments:

attr

A pointer to the `pthread_condattr_t` object from which you want to get the attribute.

pshared

A pointer to a location where the function can store the process-shared attribute. For the possible values, see [*pthread_condattr_setpshared\(\)*](#) (p. 2282).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_condattr_getpshared()* function stores, in the memory pointed to by *pshared*, the process-shared attribute from a condition variable attribute object, *attr*.

Returns:

EOK

Success.

EINVAL

Invalid condition variable attribute object specified by *attr*.

Classification:*POSIX 1003.1 THR TSH* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_condattr_init()

Initialize the attribute object for creating condition variables

Synopsis:

```
#include <pthread.h>

int pthread_condattr_init( pthread_condattr_t* attr );
```

Arguments:

attr

A pointer to the `pthread_condattr_t` object that you want to initialize.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_condattr_init()` function initializes the attributes in the condition variable attribute object `attr` to default values. Pass `attr` to [pthread_cond_init\(\)](#) (p. 2261) to define the attributes of the condition variable.

Condition variables have at least the following attributes defined:

PTHREAD_PROCESS_PRIVATE

The condition variable can be accessed only by threads created within the same process as the thread that initialized the condition variable.

PTHREAD_PROCESS_SHARED

Any thread that has access to the memory where the condition variable is allocated can access the condition variable.

For more information about these attributes, see [pthread_condattr_getpshared\(\)](#) (p. 2276) and [pthread_condattr_setpshared\(\)](#) (p. 2282).

Returns:

EOK

Success.

ENOMEM

Insufficient memory to initialize the condition variable attribute object *attr*.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_condattr_setclock()

Set the clock attribute of a condition-variable attribute object

Synopsis:

```
#include <pthread.h>

int pthread_condattr_setclock(
    pthread_condattr_t * attr,
    clockid_t id );
```

Arguments:

attr

A pointer to the `pthread_condattr_t` object for which you want to set the clock. You must have initialized this object by calling [*pthread_condattr_init\(\)*](#) (p. 2278) before calling *pthread_condattr_setclock()*.

id

A `clockid_t` object that specifies the ID of the clock that you want to use for the condition variable; one of:

- `CLOCK_REALTIME` — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- `CLOCK_SOFTTIME` — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a `CLOCK_SOFTTIME` timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, `CLOCK_SOFTTIME` behaves the same as `CLOCK_REALTIME`.

- `CLOCK_MONOTONIC` — this clock always increases at a constant rate and can't be adjusted.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_condattr_setclock()` function sets the clock attribute in an initialized attributes object referenced by `attr`. If `pthread_condattr_setclock()` is called with an `id` argument that refers to a CPU-time clock, the call fails.

The clock attribute is the clock ID of the clock that you want to use to measure the timeout service of `pthread_cond_timedwait()` (p. 2265). The default value of the clock attribute refers to the system clock.

Returns:**EOK**

Success.

EINVAL

Invalid value `attr`.

Classification:

[POSIX 1003.1 THR CS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_condattr_setpshared()

Set the process-shared attribute in a condition variable attribute object

Synopsis:

```
#include <pthread.h>

int pthread_condattr_setpshared(
    pthread_condattr_t* attr,
    int pshared );
```

Arguments:

attr

A pointer to the `pthread_condattr_t` object for which you want to set the attribute.

pshared

The new value of the process-shared attribute; one of:

- `PTHREAD_PROCESS_SHARED` — any thread that has access to the memory where the condition variable is allocated can operate on it, even if the condition variable is allocated in memory that's shared by multiple processes.
- `PTHREAD_PROCESS_PRIVATE` — the condition variable can be accessed only by threads created within the same process as the thread that initialized the condition variable; if threads from other processes try to access the `PTHREAD_PROCESS_PRIVATE` condition variable, the behavior is undefined.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_condattr_setpshared()` function sets the process-shared attribute in a condition variable attribute object, *attr* to the value given by *pshared*.

The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`.

Returns:**EOK**

Success.

EINVAL

The condition variable attribute object specified by *attr*, or the new value specified in *pshared* isn't valid.

Classification:

[POSIX 1003.1 THR TSH](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_create()

Create a thread

Synopsis:

```
#include <pthread.h>

int pthread_create( pthread_t* thread,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void* ),
                   void* arg );
```

Arguments:

thread

NULL, or a pointer to a `pthread_t` object where the function can store the thread ID of the new thread.

attr

A pointer to a `pthread_attr_t` structure that specifies the attributes of the new thread. Instead of manipulating the members of this structure directly, use [pthread_attr_init\(\)](#) (p. 2213) and the `pthread_attr_set_*` functions. For the exceptions, see “[QNX Neutrino extensions](#) (p. 2285),” below.

If `attr` is NULL, the default attributes are used (see [pthread_attr_init\(\)](#)).



If you modify the attributes in `attr` after creating the thread, the thread's attributes aren't affected.

start_routine

The routine where the thread begins, with `arg` as its only argument. If `start_routine()` returns, there's an implicit call to [pthread_exit\(\)](#) (p. 2292), using the return value of `start_routine()` as the exit status.

The thread in which `main()` was invoked behaves differently. When it returns from `main()`, there's an implicit call to [exit\(\)](#) (p. 599), using the return value of `main()` as the exit status.

arg

The argument to pass to `start_routine`.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_create()` function creates a new thread, with the attributes specified in the thread attribute object `attr`. The created thread inherits the signal mask of the parent thread, and its set of pending signals is empty.

- You must call [pthread_join\(\)](#) (p. 2303) or [pthread_detach\(\)](#) (p. 2288) for threads created with a `detachstate` attribute of `PTHREAD_CREATE_JOINABLE` (the default) before all of the resources associated with the thread can be released at thread termination.



- If you set the `__stacksize` member of `attr`, the thread's actual stack size is rounded up to a multiple of the system page size (which you can get by using the `_SC_PAGESIZE` constant in a call to [sysconf\(\)](#) (p. 3308)) if the system allocates the stack (the `__stackaddr` member of `attr` is set to `NULL`). If the stack was previously allocated by the application, its size isn't changed.

QNX Neutrino extensions

If you adhere to the POSIX standard, there are some thread attributes that you can't specify before creating the thread:

- You can't disable cancellation for a thread.
- You can't set the thread's cancellation type.
- You can't specify what happens when a signal is delivered to the thread.

There are no `pthread_attr_set_*` functions for these attributes.

As a QNX Neutrino extension, you can OR the following bits into the `__flags` member of the `pthread_attr_t` structure before calling `pthread_create()`:

PTHREAD_CANCEL_ENABLE

Cancellation requests may be acted on according to the cancellation type (the default).

PTHREAD_CANCEL_DISABLE

Cancellation requests are held pending.

PTHREAD_CANCEL_ASYNCHRONOUS

If cancellation is enabled, new or pending cancellation requests may be acted on immediately.

PTHREAD_CANCEL_DEFERRED

If cancellation is enabled, cancellation requests are held pending until a cancellation point (the default).

PTHREAD_MULTISIG_ALLOW

Terminate all the threads in the process (the POSIX default).

PTHREAD_MULTISIG_DISALLOW

Terminate only the thread that received the signal.

After creating the thread, you can change the cancellation properties by calling [*pthread_setcancelstate\(\)*](#) (p. 2392) and [*pthread_setcanceltype\(\)*](#) (p. 2394).

Returns:**EAGAIN**

Insufficient system resources to create thread.

EFAULT

An error occurred trying to access the buffers or the *start_routine* provided.

EINVAL

Invalid thread attribute object *attr*.

EOK

Success.

Examples:

Create a thread in a detached state:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* function( void* arg )
{
    printf( "This is thread %d\n", pthread_self() );
    return( 0 );
}

int main( void )
{
    pthread_attr_t attr;

    pthread_attr_init( &attr );
    pthread_attr_setdetachstate(
        &attr, PTHREAD_CREATE_DETACHED );
    pthread_create( NULL, &attr, &function, NULL );
}
```

```
/* Allow threads to run for 60 seconds. */  
sleep( 60 );  
return EXIT_SUCCESS;  
}
```

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_detach()

Detach a thread from a process

Synopsis:

```
#include <pthread.h>

int pthread_detach( pthread_t thread );
```

Arguments:

thread

The ID of the thread that you want to detach, which you can get when you call [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_detach()` function detaches the thread with the given ID from its process. When a detached thread terminates, all system resources allocated to that thread are immediately reclaimed.

Returns:

EOK

Success.

EINVAL

The thread *thread* is already detached.

ESRCH

The thread *thread* doesn't exist.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_equal()

Compare two thread IDs

Synopsis:

```
#include <pthread.h>

int pthread_equal( pthread_t t1,
                  pthread_t t2 );
```

Arguments:

t1, t2

The thread IDs that you want to compare. You can get the IDs when you call [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_equal()* function compares the thread IDs of *t1* and *t2*. It doesn't check to see if they're valid thread IDs.

Returns:

A nonzero value

The *t1* and *t2* thread IDs are equal.

0

The thread IDs aren't equal.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

pthread_exit()

Terminate a thread

Synopsis:

```
#include <pthread.h>

void pthread_exit( void* value_ptr );
```

Arguments:

value_ptr

A pointer to a value that you want to be made available to any thread joining the thread that you're terminating.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_exit()` function terminates the calling thread. If the thread is joinable, the value `value_ptr` is made available to any thread joining the terminating thread (only one thread can get the return status). If the thread is detached, all system resources allocated to the thread are immediately reclaimed.

Before the thread is terminated, any cancellation cleanup handlers that have been pushed are popped and executed, and any thread-specific-data destructor functions are executed. Thread termination doesn't implicitly release any process resources such as mutexes or file descriptors, or perform any process-cleanup actions such as calling [atexit\(\)](#) (p. 169) handlers.

An implicit call to `pthread_exit()` is made when a thread other than the thread in which `main()` was first invoked returns from the start routine that was used to create it. The return value of the start routine is used as the thread's exit status.



Don't call `pthread_exit()` from cancellation-cleanup handlers or thread-specific-data destructor functions.

For the last process thread, `pthread_exit()` behaves as if you called `exit(0)`.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_getconcurrency()

Get the level of thread concurrency

Synopsis:

```
#include <pthread.h>

int pthread_getconcurrency( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The QNX Neutrino RTOS doesn't support the multiplexing of user threads on top of several kernel scheduled entities. As such, the [pthread_setconcurrency\(\)](#) (p. 2396) and [pthread_getconcurrency\(\)](#) functions are provided for source code compatibility but they have no effect when called. To maintain the function semantics, the *new_level* parameter is saved when [pthread_setconcurrency\(\)](#) is called so that a subsequent call to [pthread_getconcurrency\(\)](#) returns the same value.

Returns:

The concurrency level set by a previous call to [pthread_setconcurrency\(\)](#) (p. 2396), or 0 if there was no previous call.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_getcpuclockid()

Return the clock ID of the CPU-time clock from a specified thread

Synopsis:

```
#include <sys/types.h>
#include <time.h>
#include <pthread.h>

int pthread_getcpuclockid( pthread_t id,
                           clockid_t* clock_id);
```

Arguments:

thread

The ID of the thread that you want to get the clock ID for, which you can get when you call [*pthread_create\(\)*](#) (p. 2284) or [*pthread_self\(\)*](#) (p. 2391).

clock_id

A pointer to a `clockid_t` object where the function can store the clock ID.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_getcpuclockid()` function gets the clock ID of the CPU-time clock of the thread specified by `id` (if the thread exists) and stores the clock ID in the object that `clock_id` points to. The CPU-time clock represents the amount of time the thread has spent actually running.

Returns:

0

Success.

ESRCH

The value specified by `id` doesn't refer to an existing thread.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>
#include <pthread.h>

int main( int argc, const char *argv[] )
{
    clockid_t clk_id;
    struct timespec tspec;
    pthread_t tid;

    tid = pthread_self();

    if (pthread_getcpuclockid( tid, &clk_id) == 0)
    {
        if (clock_gettime( clk_id, &tspec ) != 0)
        {
            perror ("clock_gettime()");
        }
        else
        {
            printf ("CPU time for tid %d is %d seconds, %ld nanoseconds.\n",
                    tid, tspec.tv_sec, tspec.tv_nsec);
        }
    }
    else
    {
        printf ("pthread_getcpuclockid(): no thread with ID %d.\n", tid);
    }

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 THR TCT (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_getname_np()

Retrieve the name of a thread

Synopsis:

```
#include <pthread.h>

int pthread_getname_np(pthread_t tid,
                       char* buffer,
                       int buffersize);
```

Arguments:

tid

The ID of the thread whose name you want to get, or 0 to get the name of the calling thread.

buffer

A storage place for the name.

buffersize

The size of the *buffer*, in bytes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_getname_np()` function retrieves the name of the specified thread as a NULL-terminated string up to a maximum size (as determined by the *buffersize*). If the thread doesn't have a name, the function stores a NULL-terminated empty string in the buffer. The “np” in the function's name stands for “non-POSIX.”



This function was added in the QNX Neutrino Core OS 6.3.2.

Returns:

EOK

Success.

EINVAL

The name buffer length is invalid or smaller than the new name length.

EPERM

You don't have the appropriate permissions to get the name.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_getschedparam()

Get a thread's scheduling parameters

Synopsis:

```
#include <pthread.h>

int pthread_getschedparam(
    const pthread_t thread,
    int *policy,
    struct sched_param *param );
```

Arguments:

thread

The ID of the thread that you want to get the scheduling parameters for. You can get a thread ID by calling [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

policy

A pointer to a location where the function can store the scheduling policy, which is one of SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, or SCHED_OTHER. As a QNX Neutrino extension, you can pass NULL for this argument.

param

A pointer to a [sched_param](#) (p. 2731) structure where the function can store the scheduling parameters.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_getschedparam()* function gets the scheduling policy and associated scheduling parameters of thread *thread* and places them in the locations that *policy* and *param* point to.

As a QNX Neutrino extension, *policy* can be NULL, in which case the function doesn't try to store the scheduling policy.



In order to get the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROCmgr_AID_SCHEDULE` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:**EOK**

Success.

EFAULT

A fault occurred trying to access the buffers provided.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

Invalid thread ID *thread*.

Classification:

[POSIX 1003.1 THR TPS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_getspecific()

Get a thread-specific data value

Synopsis:

```
#include <pthread.h>

void* pthread_getspecific( pthread_key_t key );
```

Arguments:

key

The key associated with the data that you want to get. See [pthread_key_create\(\)](#) (p. 2305).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_getspecific()* function returns the thread-specific data value currently bound to the thread-specific-data key *key* in the calling thread, or NULL if no value is bound or the key doesn't exist. You can call this function from a thread-specific-data destructor function.



You must call this function with a key that you got from *pthread_key_create()*. You can't use a key after destroying it with [pthread_key_delete\(\)](#) (p. 2308).

Returns:

The data value, or NULL.

Examples:

See [pthread_key_create\(\)](#) (p. 2305).

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_join()

Join thread

Synopsis:

```
#include <pthread.h>

int pthread_join( pthread_t thread,
                 void** value_ptr );
```

Arguments:

thread

The target thread whose termination you're waiting for.

value_ptr

NULL, or a pointer to a location where the function can store the value passed to [pthread_exit\(\)](#) (p. 2292) by the target thread.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_join()` function blocks the calling thread until the target thread *thread* terminates, unless *thread* has already terminated. If *value_ptr* is non-NULL and `pthread_join()` returns successfully, then the value passed to `pthread_exit()` by the target thread is placed in *value_ptr*. If the target thread has been canceled then *value_ptr* is set to `PTHREAD_CANCELED`.



The non-POSIX [pthread_timedjoin\(\)](#) (p. 2433) function is similar to `pthread_join()`, except that an error is returned if the join doesn't occur before a given time.

The target thread must be joinable. Multiple `pthread_join()`, `pthread_timedjoin()`, `ThreadJoin()`, and `ThreadJoin_r()` calls on the same target thread aren't allowed. When `pthread_join()` returns successfully, the target thread has been terminated.

Returns:

EOK

Success.

EBUSY

The thread *thread* is already being joined.

EDEADLK

The thread *thread* is the calling thread.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

The thread *thread* isn't joinable.

ESRCH

The thread *thread* doesn't exist.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_key_create()

Create a thread-specific data key

Synopsis:

```
#include <pthread.h>

int pthread_key_create( pthread_key_t * key,
                       void (*destructor)( void * ) );
```

Arguments:

key

A pointer to a pthread_key_t object where the function can store the new key.

destructor

NULL, or a function to be called when you destroy the key.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_key_create()* function creates a thread-specific data key that's available to all threads in the process and binds an optional destructor function *destructor* to the key. If the function completes successfully the new key is returned in *key*.



Although the same key may be used by different threads, the values bound to the key using *pthread_setspecific()* are maintained on a per-thread basis and persist only for the lifetime of the thread.

When you create a key, the value NULL is bound with the key in all active threads. When you create a thread, the value NULL is bound to all defined keys in the new thread.

You can optionally associate a destructor function with each key value. At thread exit, if the key has a non-NULL destructor pointer, and the thread has a non-NULL value bound to the key, the destructor function is called with the bound value as its only

argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all destructor functions have been called for a thread, there are still non-NULL bound values, the destructor function is called repeatedly a maximum of `PTHREAD_DESTRUCTOR_ITERATIONS` times for each non-NULL bound value.

Returns:

EOK

Success.

EAGAIN

Insufficient system resources to create key or `PTHREAD_KEYS_MAX` has been exceeded.

ENOMEM

Insufficient memory to create key.

Examples:

This example shows how you can use thread-specific data to provide per-thread data in a thread-safe function:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

pthread_key_t buffer_key;

void buffer_key_destruct( void *value )
{
    free( value );
    pthread_setspecific( buffer_key, NULL );
}

char *lookup( void )
{
    char *string;

    string = (char *)pthread_getspecific( buffer_key );
    if( string == NULL ) {
        string = (char *) malloc( 32 );
        sprintf( string, "This is thread %d\n", pthread_self() );
        pthread_setspecific( buffer_key, (void *)string );
    }

    return( string );
}

void *function( void *arg )
{
    while( 1 ) {
        puts( lookup() );
    }

    return( 0 );
}

int main( void )
{

```

```

pthread_key_create( &buffer_key,
                   &buffer_key_destruct );
pthread_create( NULL, NULL, &function, NULL );

/* Let the threads run for 60 seconds. */
sleep( 60 );

return EXIT_SUCCESS;
}

```

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The *pthread_key_create()* function is part of the POSIX 1003.1-1996 draft standard; its specific behavior may vary from system to system.

Before each destructor is called, the thread's value for the corresponding key is set to NULL. Calling:

```
pthread_setspecific( key, NULL );
```

in a key destructor isn't required; this lets you use the same destructor for several keys.

pthread_key_delete()

Delete a thread-specific data key

Synopsis:

```
#include <pthread.h>

int pthread_key_delete( pthread_key_t key );
```

Arguments:

key

The key, which you created by calling [pthread_key_create\(\)](#) (p. 2305), that you want to delete.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_key_delete()` function deletes the thread-specific data key `key` that you previously created with `pthread_key_create()`. The destructor function bound to `key` isn't called by this function, and won't be called at thread termination. You can call this function from a thread specific data destructor function.

If you need to release any data bound to the key in any threads, do so before deleting the key.

Returns:

EOK

Success.

EINVAL

Invalid thread-specific data key `key`.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_kill()

Send a signal to a thread

Synopsis:

```
#include <signal.h>

int pthread_kill( pthread_t thread,
                  int sig );
```

Arguments:

thread

The ID of the thread that you want to send the signal to, which you can get when you call [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

sig

The signal that you want to send, or 0 if you just want to check for errors.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_kill()* function sends the signal *sig* to the thread *thread*. The target thread and calling thread must be in the same process. If *sig* is zero, error checking is performed but no signal is sent.

In order to send signals to a process with a different real or effective user ID, the calling process must have the `PROCmgr_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

EOK

Success.

EAGAIN

Insufficient system resources are available to deliver the signal.

EINVAL

Invalid signal number *sig*.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

Invalid thread ID *thread*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_destroy()

Destroy a mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutex_destroy( pthread_mutex_t* mutex );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_destroy()` function destroys the unlocked mutex *mutex*.

You can destroy a locked mutex only if you're the owner of that mutex.



Once you've destroyed a mutex, don't reuse it without reinitializing it by calling [pthread_mutex_init\(\)](#) (p. 2316).

Returns:

EOK

Success.

EBUSY

The *mutex* is locked by another thread.

EINVAL

Invalid mutex *mutex*.

Classification:*POSIX 1003.1 THR* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_getprioceiling()

Get a mutex's priority ceiling

Synopsis:

```
#include <pthread.h>

int pthread_mutex_getprioceiling(
    const pthread_mutex_t* mutex,
    int* prioceiling );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object that you want to priority ceiling for.

prioceiling

A pointer to a location where the function can store the priority ceiling.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_getprioceiling()` function returns the priority ceiling of *mutex* in *prioceiling*.

Returns:

EOK

Success.

EINVAL

The mutex specified by *mutex* doesn't currently exist.

EPERM

The calling thread doesn't have permission to get the priority ceiling.

Classification:*POSIX 1003.1 THR TPP* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_init()

Initialize mutex

Synopsis:

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(
    pthread_mutex_t* mutex,
    const pthread_mutexattr_t* attr );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object that you want to initialize.

attr

NULL, or a pointer to a `pthread_mutexattr_t` object that specifies the attributes that you want to use for the mutex. For more information, see [pthread_mutexattr_init\(\)](#) (p. 2347).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_init()` function initializes the given mutex object, using the attributes specified by the mutex attributes object `attr`. If `attr` is NULL, then the mutex is initialized with the default attributes (see [pthread_mutexattr_init\(\)](#) (p. 2347)). After initialization, the mutex is in an unlocked state.



You should allocate synchronization objects only in normal memory mappings.

On certain processors, atomic operations such as calls to [pthread_mutex_lock\(\)](#) (p. 2318) will cause a fault if the control structure is allocated in uncached memory.

You can initialize a statically allocated mutex with the default attributes by assigning to it the macro `PTHREAD_MUTEX_INITIALIZER` or `PTHREAD_RMUTEX_INITIALIZER` (for recursive mutexes).



To destroy a mutex, call [pthread_mutex_destroy\(\)](#) (p. 2312). Once you've destroyed a mutex, don't reuse it without reinitializing it by calling `pthread_mutex_init()`.

Returns:

EOK

Success.

EAGAIN

All kernel synchronization objects are in use.

EBUSY

The given mutex was previously initialized and hasn't been destroyed.

EFAULT

A fault occurred when the kernel tried to access *mutex* or *attr*.

EINVAL

The value specified by *attr* is invalid.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_lock()

Lock a mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutex_lock( pthread_mutex_t* mutex );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object that you want to lock.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_lock()` function locks the mutex object referenced by *mutex*. If the mutex is already locked, then the calling thread blocks until it has acquired the mutex. When the function returns, the mutex object is locked and owned by the calling thread.

This function's behavior when you try to lock a mutex that you already own depends on the type of the mutex. For more information, see the entry for [`pthread\_mutexattr\_settype\(\)`](#) (p. 2357).

By default, if a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the higher-priority blocked thread waiting for the mutex. The owner returns to its real priority when it unlocks the mutex. For more information, see “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

If the mutex is recursive, you must call [`pthread\_mutex\_unlock\(\)`](#) (p. 2329) for each corresponding call to lock the mutex.

If a signal is delivered to a thread that's waiting for a mutex, the thread resumes waiting for the mutex on returning from the signal handler.

If, before initializing the mutex, you've called [pthread_mutexattr_setwakeup_np\(\)](#) (p. 2359) to enable wake-ups, you can later call [pthread_mutex_wakeup_np\(\)](#) (p. 2331), to wake up any threads that are blocked on the mutex. The “np” in these functions' names stands for “non-POSIX.”

Returns:

EOK

Success.

EAGAIN

Insufficient system resources available to lock the mutex.

EDEADLK

The calling thread already owns *mutex*, and the mutex doesn't allow recursive behavior.

EFAULT

A fault occurred when the kernel tried to access the buffers you provided.

EINTR

(QNX Neutrino extension) The thread was unblocked by a call to [pthread_mutex_wakeup_np\(\)](#) (p. 2331).

EINVAL

The mutex is invalid, or it has died (see [SyncMutexEvent\(\)](#) (p. 3291)).

ETIMEDOUT

A kernel timeout unblocked the call.

Examples:

This example shows how you can use a mutex to synchronize access to a shared variable. In this example, *function1()* and *function2()* both attempt to access and modify the global variable *count*. Either thread could be interrupted between modifying *count* and assigning its value to the local *tmp* variable. Locking *mutex* prevents this from happening; if one thread has *mutex* locked, the other thread waits until it's unlocked, before continuing.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int count = 0;
```

```

void* function1( void* arg )
{
    int tmp = 0, ret_code;

    while( 1 ) {
        ret_code = pthread_mutex_lock( &mutex );
        if (ret_code == EOK)
        {
            tmp = count++;
            ret_code = pthread_mutex_unlock( &mutex );
            if (ret_code != EOK)
            {
                printf ("pthread_mutex_unlock() failed: %s\n",
                    strerror(ret_code));
            }

            printf ("Count is %d\n", tmp );
        }
        else
        {
            printf ("pthread_mutex_lock() failed: %s\n",
                strerror(ret_code));
        }

        /* Snooze for 1 second */
        sleep( 1 );
    }

    return 0;
}

void* function2( void* arg )
{
    int tmp = 0, ret_code;

    while( 1 ) {
        ret_code = pthread_mutex_lock( &mutex );
        if (ret_code == EOK)
        {
            tmp = count--;
            ret_code = pthread_mutex_unlock( &mutex );
            if (ret_code != EOK)
            {
                printf ("** pthread_mutex_unlock() failed: %s\n",
                    strerror (ret_code));
            }
            printf ("** Count is %d\n", tmp );
        }
        else
        {
            printf ("** pthread_mutex_lock() failed: %s\n",
                strerror (ret_code));
        }

        /* Snooze for 2 seconds */
        sleep( 2 );
    }

    return 0;
}

int main( void )
{
    int ret_code;

    ret_code = pthread_create( NULL, NULL, &function1, NULL );
    if (ret_code != EOK)
    {
        printf ("Couldn't create first thread: %s\n",
            strerror (ret_code));
    }
    ret_code = pthread_create( NULL, NULL, &function2, NULL );
    if (ret_code != EOK)
    {
        printf ("Couldn't create second thread: %s\n",
            strerror (ret_code));
    }

    /* Let the threads run for 60 seconds. */
    sleep( 60 );
}

```

```
    return EXIT_SUCCESS;  
}
```

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_setprioceiling()

Set a mutex's priority ceiling

Synopsis:

```
#include <pthread.h>

int pthread_mutex_setprioceiling(
    pthread_mutex_t* mutex,
    int prioceiling,
    int* old_ceiling );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object that you want to priority ceiling for.

prioceiling

The new value for the priority ceiling.

old_ceiling

A pointer to a location where the function can store the old value.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



In order to change the priority ceiling to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [*procmgr_ability\(\)*](#) (p. 2141).

Description:

The `pthread_mutex_setprioceiling()` function locks *mutex* (or blocks until it can lock it), changes its priority ceiling to *prioceiling*, and releases it. When the change is successful, the previous priority ceiling is returned in *old_ceiling*.

Returns:

EOK

Success.

EINVAL

The mutex specified by *mutex* doesn't currently exist, or the priority requested by *prioceiling* is out of range.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 THR TPP](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_timedlock(), pthread_mutex_timedlock_monotonic()

Attempt to lock a mutex

Synopsis:

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(
    pthread_mutex_t * mutex,
    const struct timespec * abs_timeout );

int pthread_mutex_timedlock_monotonic(
    pthread_mutex_t * mutex,
    const struct timespec * abs_timeout );
```

Arguments:

mutex

The mutex that you want to lock.

abs_timeout

A pointer to a [timespec](#) (p. 3473) structure that specifies the maximum time to wait to lock the mutex, expressed as an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_mutex_timedlock()* function locks the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available as in the [pthread_mutex_lock\(\)](#) (p. 2318) function. If the mutex can't be locked without waiting for another thread to unlock the mutex, the wait is terminated when the specified timeout expires.

The *pthread_mutex_timedlock_monotonic()* function is a QNX Neutrino extension; it's similar to *pthread_mutex_timedlock()*, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

By default, if a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the

higher-priority blocked thread waiting for the mutex. The owner returns to its real priority when it unlocks the mutex. For more information, see “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (i.e., when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The timeout is based on the `CLOCK_REALTIME` clock. The `timespec` datatype is defined in the `<time.h>` header.

If the mutex can be locked immediately, the validity of the *abs_timeout* parameter isn't checked, and the function won't fail with a timeout.

As a consequence of the priority inheritance rules (for mutexes initialized with the `PTHREAD_PRIO_INHERIT` protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex is adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

Returns:

Zero on success, or an error number to indicate the error.

Errors:**EAGAIN**

The mutex couldn't be acquired because the maximum number of recursive locks for the mutex has been exceeded.

EDEADLK

The current thread already owns the mutex.

EINVAL

One of the following occurred:

- The mutex was created with a protocol attribute of `PTHREAD_PRIO_PROTECT`, and the calling thread's priority is higher than the mutex's current priority ceiling.
- The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million
- The value specified by *mutex* doesn't refer to an initialized mutex object.
- The mutex has died; see [SyncMutexEvent\(\)](#) (p. 3291).

EOWNERDEAD

The owner of the lock died while holding it.

ETIMEDOUT

The mutex couldn't be locked before the specified timeout expired.

Classification:

pthread_mutex_timedlock() is [POSIX 1003.1 THR TMO](#) (p. 46);

pthread_mutex_timedlock_monotonic() is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_trylock()

Attempt to lock a mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutex_trylock( pthread_mutex_t* mutex );
```

Arguments:

mutex

A pointer to the pthread_mutex_t object that you want to try to lock.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_mutex_trylock()* function attempts to lock the mutex *mutex*, but doesn't block the calling thread if the mutex is already locked.

Returns:

EOK

Success.

EAGAIN

Insufficient resources available to lock the mutex.

EBUSY

The *mutex* was already locked.

EINVAL

The mutex is invalid, or it has died (see [SyncMutexEvent\(\)](#) (p. 3291)).

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_unlock()

Unlock mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object that you want to unlock.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_unlock()` function unlocks the mutex *mutex*. The mutex should be owned by the calling thread. If there are threads blocked on the mutex then the highest priority waiting thread is unblocked and becomes the next owner of the mutex.

If the thread's priority was raised when a higher-priority thread attempted to lock the mutex, the calling thread's priority is again adjusted when it unlocks the mutex; its new priority is the maximum of its own priority and the priorities of those threads it still blocks, either directly or indirectly. For more information, see “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

If *mutex* has been locked more than once, it must be unlocked the same number of times before the next thread is given ownership of the mutex. This only works for recursive mutexes.

Returns:

EOK

Success.

EINVAL

Invalid mutex *mutex*.

EPERM

Current thread doesn't own *mutex*.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutex_wakeup_np()

Unblock any threads that are waiting on a mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutex_wakeup_np( pthread_mutex_t * mutex,
                             pid_t pid,
                             pthread_t tid );
```

Arguments:

mutex

A pointer to the `pthread_mutex_t` object for which you want to wake up any waiting threads.

pid

The ID of a specific process whose thread or threads you want to unblock, or 0 to unblock a specific thread in the current process (if *tid* isn't 0) or all threads for all processes (if *tid* is 0).

tid

The ID of a thread in the specified process that you want to unblock, or 0 to unblock all threads.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_wakeup_np()` function unblocks any threads that are waiting on the specified mutex, provided that `PTHREAD_WAKEUP_ENABLE` is set in the mutex attributes (see [pthread_mutexattr_setwakeup_np\(\)](#) (p. 2359)). The “np” in these functions' names stands for “non-POSIX.”

The waiting threads' call to [pthread_mutex_lock\(\)](#) (p. 2318) for returns with an error code of `EINTR`, which is a non-POSIX return code for this function.

Returns:

EOK

Success.

EINVAL

One of the following:

- The mutex doesn't exist
- The PTHREAD_WAKEUP_ENABLE flag wasn't set for the mutex.
- You gave a specific thread ID, and that thread isn't blocked on the mutex.

ESRCH

The specified process and thread couldn't be found.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_destroy()

Destroy mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_destroy(
    pthread_mutexattr_t* attr );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_destroy()` function destroys the mutex attribute object *attr*.



Once you've destroyed a mutex attribute object, don't reuse it without reinitializing it by calling [pthread_mutexattr_init\(\)](#) (p. 2347).

Returns:

EOK

Success.

EINVAL

Invalid mutex attribute object *attr*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_getprioceiling()

Get the priority ceiling of a mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t* attr,
    int* prioceiling );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.

prioceiling

A pointer to a location where the function can store the priority ceiling.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getprioceiling()` function sets *prioceiling* to the current mutex attribute *attr*'s scheduling priority ceiling. The mutex attribute object *attr* must have been previously created with `pthread_mutexattr_init()`.

Returns:

EOK

Success.

EINVAL

Invalid value specified by *attr* or *prioceiling*.

EPERM

The caller doesn't have the privilege to perform the operation.

Classification:

POSIX 1003.1 THR TPP (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_getprotocol()

Get a mutex's scheduling protocol

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(
    pthread_mutexattr * attr,
    int * protocol );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.

protocol

A pointer to a location where the function can store the scheduling protocol.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getprotocol()` function sets *protocol* to the current mutex attribute *attr*'s scheduling protocol. The structure pointed to by *attr* must have been previously created with `pthread_mutexattr_init()`.

The *protocol* attribute defines the protocol for using mutexes: Currently, *protocol* may be set to:

PTHREAD_PRIO_INHERIT

When a thread is blocking higher-priority threads by locking one or more mutexes with this attribute, the thread's priority is raised to that of the highest priority thread waiting on the `PTHREAD_PRIO_INHERIT` mutex.

PTHREAD_PRIO_PROTECT

The thread executes at the highest priority or priority ceilings of all the mutexes owned by the thread and initialized with `PTHREAD_PRIO_PROTECT`, whether other threads are blocked or not.

PTHREAD_PRIO_NONE

Owning the mutex doesn't affect the priority and scheduling of the thread.

A thread holding a `PTHREAD_PRIO_INHERIT` mutex won't be moved to the tail of the scheduling queue if its original priority is changed (by a call to *pthread_schedsetparam()*, for example). This remains true if the thread unlocks the `PTHREAD_PRIO_INHERIT` mutex.

Returns:

EOK

Success.

EINVAL

Invalid mutex attribute *attr*.

Classification:

[*POSIX 1003.1 THR TPP/TPI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_getpshared()

Get the process-shared attribute from a mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_getpshared(
    const pthread_mutexattr_t* attr,
    int* pshared );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.

pshared

A pointer to a location where the function can store the process-shared attribute.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getpshared()` function gets the process-shared attribute from the mutex attribute object `attr` and stores it in the memory pointed to by `pshared`.

If the process-shared attribute is set to `PTHREAD_PROCESS_SHARED`, any thread that has access to the memory where the condition variable is allocated can operate on it, even if the condition variable is allocated in memory that's shared by multiple processes.

If the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`, the condition variable can only be accessed by threads created within the same process as the thread that initialized the condition variable; if threads from other processes try to access the `PTHREAD_PROCESS_PRIVATE` condition variable, the behavior is undefined. The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`.

Returns:**EOK**

Success.

EINVAL

The mutex attribute object specified by *attr* is invalid.

Classification:

POSIX 1003.1 THR TSH (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_getrecursive()

Get the recursive attribute from a mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_getrecursive(
    const pthread_mutexattr_t* attr,
    int* recursive );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.

pshared

A pointer to a location where the function can store the recursive attribute.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getrecursive()` function gets the recursive attribute from the mutex attribute object *attr* and stores it in *recursive*.

If the recursive attribute is set to `PTHREAD_RECURSIVE_ENABLE`, a thread that has already locked the mutex can lock it again without blocking. If the recursive attribute is set to `PTHREAD_RECURSIVE_DISABLE`, any thread that tries to lock the mutex will block, if that mutex is already locked.

The default value of the recursive attribute is `PTHREAD_RECURSIVE_DISABLE`.

Returns:

EOK

Success.

EINVAL

Invalid mutex attribute object *attr*.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_gettype()

Get a mutex type

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_gettype(
    const pthread_mutexattr_t * attr,
    int * type );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.

type

A pointer to a location where the function can store the type.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_mutexattr_gettype()* function gets the mutex type from the mutex attribute object *attr* and stores it in *type*. Valid mutex types include:

PTHREAD_MUTEX_NORMAL

No deadlock detection. A thread that attempts to relock this mutex without first unlocking it deadlocks. Attempts to unlock a mutex locked by a different thread or attempts to unlock an unlocked mutex result in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK

Provides error checking. A thread returns with an error when it attempts to:

- Relock this mutex without first unlocking it.
- Unlock a mutex that another thread has locked.
- Unlock an unlocked mutex.

PTHREAD_MUTEX_RECURSIVE

A thread that attempts to relock this mutex without first unlocking it succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type PTHREAD_MUTEX_NORMAL can't occur with this mutex type. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread that attempts to unlock a mutex that another thread has locked, or unlock an unlocked mutex, returns with an error.

PTHREAD_MUTEX_DEFAULT

The default value of the *type* attribute. Attempts to recursively lock a mutex of this type, or unlock a mutex of this type that isn't locked by the calling thread, or unlock a mutex of this type that isn't locked, results in undefined behavior.

Returns:

EOK

Success.

EINVAL

Invalid mutex attribute object *attr*.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

An application shouldn't use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a [pthread_cond_wait\(\)](#) (p. 2269) or [pthread_cond_timedwait\(\)](#) (p. 2265) may not actually release the mutex (if it's been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

pthread_mutexattr_getwakeup_np()

Get the wake-up attribute of a mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_getwakeup_np(
    const pthread_mutexattr_t *attr,
    int *wakeup );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.

wakeup

A pointer to a location where the function can store the current value of the attribute.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getwakeup_np()` function gets the value of the wake-up attribute for the given mutex. If this attribute is enabled, a thread can call [pthread_mutex_wakeup_np\(\)](#) (p. 2331) to unblock any threads that are waiting for the mutex.

The “np” in these functions' names stands for “non-POSIX.”

Returns:

`EOK`

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

pthread_mutexattr_init()

Initialize a mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_init( pthread_mutexattr_t* attr );
```

Arguments:

attr

A pointer to the pthread_mutexattr_t object that you want to initialize.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_mutexattr_init()* function initializes the attributes in the mutex attribute object *attr* to their default values. After initializing a mutex attribute object, you can use it to initialize one or more mutexes by calling [pthread_mutex_init\(\)](#) (p. 2316).

The mutex attributes and their default values are:

__protocol

PTHREAD_PRIO_INHERIT—when a thread is blocking higher-priority threads by locking one or more mutexes with this attribute, the thread's priority is raised to that of the highest priority thread waiting on the PTHREAD_PRIO_INHERIT mutex.

__recursive

PTHREAD_RECURSIVE_DISABLE—threads can't recursively lock a mutex; any thread that tries to lock an already locked mutex becomes blocked.

After calling this function, you can use the *pthread_mutexattr_** family of functions to make any changes to the attributes:

Get	Set
<i>pthread_mutexattr_getprioceiling()</i> (p. 2335)	<i>pthread_mutexattr_setprioceiling()</i> (p. 2349)
<i>pthread_mutexattr_getprotocol()</i> (p. 2337)	<i>pthread_mutexattr_setprotocol()</i> (p. 2351)
<i>pthread_mutexattr_getpshared()</i> (p. 2339)	<i>pthread_mutexattr_setpshared()</i> (p. 2353)
<i>pthread_mutexattr_getrecursive()</i> (p. 2341)	<i>pthread_mutexattr_setrecursive()</i> (p. 2355)
<i>pthread_mutexattr_gettype()</i> (p. 2343)	<i>pthread_mutexattr_settype()</i> (p. 2357)
<i>pthread_mutexattr_getwakeup_np()</i> (p. 2345)	<i>pthread_mutexattr_setwakeup_np()</i> (p. 2359)

Returns:**EOK**

Success.

ENOMEM

Insufficient memory to initialize mutex attribute object *attr*.

Classification:

[*POSIX 1003.1 THR*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_setprioceiling()

Set the priority ceiling of a mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(
    pthread_mutexattr_t* attr,
    int prioceiling );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

prioceiling

The new value for the priority ceiling.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_setprioceiling()` function sets the mutex attribute *attr*'s scheduling priority ceiling to *prioceiling*. Note that *attr* must have been previously created with `pthread_mutexattr_init()`.



In order to change the priority ceiling to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

EOK

Success.

EINVAL

Invalid value specified by *attr* or *prioceiling*.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 THR TPP](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_setprotocol()

Set a mutex's scheduling protocol

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(
    pthread_mutexattr * attr,
    int protocol );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

protocol

The new value of the scheduling protocol; one of:

- `PTHREAD_PRIO_INHERIT` — when a thread is blocking higher-priority threads by locking one or more mutexes with this attribute, raise the thread's priority to that of the highest priority thread waiting on the `PTHREAD_PRIO_INHERIT` mutex.
- `PTHREAD_PRIO_PROTECT` — execute the thread at the highest priority or priority ceilings of all the mutexes owned by the thread and initialized with `PTHREAD_PRIO_PROTECT`, whether other threads are blocked or not.



In order to change the priority ceiling to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

- `PTHREAD_PRIO_NONE` — owning the mutex doesn't affect the priority and scheduling of the thread.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_setprotocol()` function sets the mutex attribute *attr*'s scheduling protocol to *protocol*. The structure pointed to by *attr* must have been previously created with `pthread_mutexattr_init()`.

The *protocol* attribute defines the protocol for using mutexes. A thread holding a `PTHREAD_PRIO_INHERIT` mutex won't be moved to the tail of the scheduling queue if its original priority is changed (by a call to `pthread_setschedparam()`, for example). This remains true if the thread unlocks the `PTHREAD_PRIO_INHERIT` mutex.

Returns:**EOK**

Success.

ENOTSUP

The *protocol* argument is an unsupported or an invalid value.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 THR TPPITPI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_setpshared()

Set the process-shared attribute in mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_setpshared(
    pthread_mutexattr_t* attr,
    int pshared );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

pshared

The new value of the process-shared attribute; one of:

- `PTHREAD_PROCESS_SHARED` — any thread that has access to the memory where the mutex is allocated can operate on it, even if the mutex is allocated in memory that's shared by multiple processes.
- `PTHREAD_PROCESS_PRIVATE` — the mutex can be accessed only by threads created within the same process as the thread that initialized the mutex; if threads from other processes try to access the `PTHREAD_PROCESS_PRIVATE` mutex, the behavior is undefined.

The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_setpshared()` function sets the process-shared attribute in a mutex attribute object, `attr`, to the value given by `pshared`.

Returns:

EOK

Success.

EINVAL

Invalid mutex attribute object, *attr*.

EINVAL

The new value specified in *pshared* isn't PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

Classification:

POSIX 1003.1 THR TSH (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_setrecursive()

Set the recursive attribute in mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_setrecursive(
    pthread_mutexattr_t* attr,
    int recursive );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

recursive

The new value for the recursive attribute; one of:

- `PTHREAD_RECURSIVE_ENABLE` — a thread that has already locked the mutex can lock it again without blocking.
- `PTHREAD_RECURSIVE_DISABLE` — any thread that tries to lock the mutex blocks, if that mutex is already locked.

The default value of the recursive attribute is `PTHREAD_RECURSIVE_DISABLE`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_setrecursive()` function sets the recursive attribute in the mutex attribute object `attr` to the value given by `recursive`.



The [pthread_mutexattr_settype\(\)](#) (p. 2357) function provides a more portable way to make a mutex recursive.

Returns:**EOK**

Success.

EINVAL

Invalid mutex attribute object, *attr*, or the value specified by *recursive* isn't PTHREAD_RECURSIVE_ENABLE or PTHREAD_RECURSIVE_DISABLE.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_mutexattr_settype()

Set a mutex type

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_settype(
    pthread_mutexattr_t * attr,
    int type );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

type

The new type; one of:

- `PTHREAD_MUTEX_NORMAL` — no deadlock detection. A thread that attempts to relock this mutex without first unlocking it deadlocks. Attempts to unlock a mutex locked by a different thread or attempts to unlock an unlocked mutex result in undefined behavior.
- `PTHREAD_MUTEX_ERRORCHECK` — provides error checking. A thread returns with an error when it attempts to relock this mutex without first unlocking it, unlock a mutex that another thread has locked, or unlock an unlocked mutex.
- `PTHREAD_MUTEX_RECURSIVE` — a thread that attempts to relock this mutex without first unlocking it succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type `PTHREAD_MUTEX_NORMAL` can't occur with this mutex type. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread that attempts to unlock a mutex that another thread has locked, or unlock an unlocked mutex, returns with an error.
- `PTHREAD_MUTEX_DEFAULT` — the default value of the *type* attribute. Attempts to recursively lock a mutex of this type, or unlock a mutex of this type that isn't locked by the calling thread, or unlock a mutex of this type that isn't locked, results in undefined behavior.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_settype()` function sets the mutex type in the mutex attribute object *attr* to the value given by *type*.

Returns:

EOK

Success.

EINVAL

The value specified by *attr* or *type* is invalid.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

An application shouldn't use a `PTHREAD_MUTEX_RECURSIVE` mutex with condition variables because the implicit unlock performed for a [pthread_cond_wait\(\)](#) (p. 2269) or [pthread_cond_timedwait\(\)](#) (p. 2265) may not actually release the mutex (if it's been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

pthread_mutexattr_setwakeup_np()

Set the wake-up attribute of a mutex

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_setwakeup_np(
    pthread_mutexattr_t *attr,
    int wakeup );
```

Arguments:

attr

A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

wakeup

The new value for the attribute; one of:

- `PTHREAD_WAKEUP_DISABLE`
- `PTHREAD_WAKEUP_ENABLE`

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_setwakeup_np()` function sets the wake-up attribute of the given mutex. If this attribute is enabled, a thread can call [*pthread_mutex_wakeup_np\(\)*](#) (p. 2331) to unblock any threads that are waiting for the mutex.

The “np” in these functions' names stands for “non-POSIX.”

Returns:

`EOK`

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

pthread_once()

Dynamic package initialization

Synopsis:

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once( pthread_once_t* once_control,
                 void (*init_routine)(void) );
```

Arguments:

once_control

A pointer to a pthread_once_t object that the function uses to determine whether or not to run the initialization code.



You must set the pthread_once_t object to the macro PTHREAD_ONCE_INIT before using it for the first time.

init_routine

The function that you want to call to do any required initialization.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_once()* function uses the once-control object *once_control* to determine whether the initialization routine *init_routine* should be called.



You should allocate the control object only in normal memory mappings. Doing otherwise will cause a fault on certain processors.

The first call to *pthread_once()* by any thread in a process, with a given *once_control*, calls *init_routine* with no arguments. Subsequent calls of *pthread_once()* with the same *once_control* won't call *init_routine*.



No thread will execute past this function until the *init_routine* returns.

Returns:

EOK

Success.

EINVAL

Uninitialized once-control object *once_control*.

Examples:

This example shows how you can use once-initialization to initialize a library; both *library_entry_point1()* and *library_entry_point2()* need to initialize the library, but that needs to happen only once:

```
#include <stdio.h>
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

void library_init( void )
{
    /* initialize the library */
}

void library_entry_point1( void )
{
    pthread_once( &once_control, library_init );
    /* do stuff for library_entry_point1... */
}

void library_entry_point2( void )
{
    pthread_once( &once_control, library_init );
    /* do stuff for library_entry_point1... */
}
```

This initializes the library once; if multiple threads call *pthread_once()*, only one actually enters the *library_init()* function. The other threads block at the *pthread_once()* call until *library_init()* has returned. The *pthread_once()* function also ensures that *library_init()* is only ever called once; subsequent calls to the library entry points skip the call to *library_init()*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

pthread_rwlock_destroy()

Destroy a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_destroy( pthread_rwlock_t* rwl );
```

Arguments:

rwl

A pointer to a `pthread_rwlock_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlock_destroy()` function destroys the read-write lock referenced by *rwl*, and releases the system resources used by the lock. You can destroy the read-write lock if one of the following is true:

- no thread has a active shared or exclusive lock on *rwl*
- the calling thread has an active exclusive lock on *rwl*.



After successfully destroying a read-write lock, don't use it again without reinitializing it by calling [`pthread\_rwlock\_init\(\)`](#) (p. 2366).

Returns:

EOK

Success.

EBUSY

The read-write lock *rwl* is still in use. The calling thread doesn't have an exclusive lock.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_init()

Initialize a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_init(
    pthread_rwlock_t * rwlock,
    const pthread_rwlockattr_t * attr );
```

Arguments:

rwlock

A pointer to a `pthread_rwlock_t` object that you want to initialize.

attr

NULL, or a pointer to a `pthread_rwlockattr_t` object that specifies the attributes you want to use for the read-write lock; see [pthread_rwlockattr_init\(\)](#) (p. 2387).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlock_init()` function initializes the read-write lock referenced by *rwlock* with the attributes of *attr*. You must initialize read-write locks before using them. If *attr* is NULL, *rwlock* is initialized with the default values for the attributes.



You should allocate synchronization objects only in normal memory mappings. On certain processors, atomic operations such as locking will cause a fault if the control structure is allocated in uncached memory.

Following a successful call to `pthread_rwlock_init()`, the read-write lock is unlocked, and you can use it in subsequent calls to [pthread_rwlock_destroy\(\)](#) (p. 2364), [pthread_rwlock_rdlock\(\)](#) (p. 2369), [pthread_rwlock_tryrdlock\(\)](#) (p. 2375), [pthread_rwlock_trywrlock\(\)](#) (p. 2377), and [pthread_rwlock_wrlock\(\)](#) (p. 2381). This lock remains usable until you destroy it by calling `pthread_rwlock_destroy()`.

If the read-write lock is statically allocated, you can initialize it with the default values by setting it to `PTHREAD_RWLOCK_INITIALIZER`.

More than one thread may hold a shared lock at any time, but only one thread may hold an exclusive lock. This avoids reader and writer starvation during frequent contention by:

- favoring blocked readers over writers after a writer has just released an exclusive lock, and
- favoring writers over readers when there are no blocked readers.

Under heavy contention, the lock alternates between a single exclusive lock followed by a batch of shared locks.

Returns:

EOK

Success.

EAGAIN

Insufficient system resources to initialize the read-write lock.

EBUSY

The read-write lock *rwl* has been initialized or unsuccessfully destroyed.

EFAULT

A fault occurred when the kernel tried to access *rwl* or *attr*.

EINVAL

Invalid read-write lock attribute object *attr*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Beware of *priority inversion* when using read-write locks. A high-priority thread may be blocked waiting on a read-write lock locked by a low-priority thread.

The microkernel has no knowledge of read-write locks, and therefore can't boost the low-priority thread to prevent the priority inversion.

pthread_rwlock_rdlock()

Acquire a shared read lock on a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_rdlock( pthread_rwlock_t* rwlock );
```

Arguments:

rwlock

A pointer to a pthread_rwlock_t object that you want to lock for reading.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_rwlock_rdlock()* function acquires a shared lock on the read-write lock referenced by *rwlock*. If the read-write lock is already exclusively locked, the calling thread blocks until the exclusive lock is released.



If another thread is currently waiting to acquire the read-write lock for writing, *pthread_rwlock_rdlock()* blocks. This can cause a deadlock if a thread attempts to recursively acquire the lock for reading, while another thread is trying to acquire the lock for writing.

If a signal is delivered to a thread waiting to lock a read-write lock, it will resume waiting for the lock after returning from the signal handler.

A thread may hold several read locks on the same read-write lock; it must call [*pthread_rwlock_unlock\(\)*](#) (p. 2379) multiple times to release its read lock.

Returns:

EOK

Success.

EAGAIN

On the first use of statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.

EDEADLK

The calling thread already has an exclusive lock for *rw/*.

EFAULT

A fault occurred when the kernel tried to access *rw/*.

EINVAL

The read-write lock *rw/* is invalid.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_timedrdlock()

Lock a read-write lock for reading

Synopsis:

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(
    pthread_rwlock_t * rwlock,
    const struct timespec * abs_timeout );
```

Arguments:

rwlock

The read-write lock that you want to lock.

abs_timeout

A pointer to a [timespec](#) (p. 3473) that specifies the maximum time to wait to acquire the lock, expressed as an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlock_timedrdlock()` function applies a read lock to the read-write lock referenced by *rwlock* as in [pthread_rwlock_rdlock\(\)](#) (p. 2369).

However, if the lock can't be acquired without waiting for other threads to unlock it, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (i.e. when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The timeout is based on the `CLOCK_REALTIME` clock.

If the read-write lock can be locked immediately, the validity of the *abs_timeout* parameter isn't checked, and the function won't fail with a timeout.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to *pthread_rwlock_timedrdlock()*, upon return from the signal handler the thread resumes waiting for the lock as if it hadn't been interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*. The results are undefined if this function is called with an uninitialized read-write lock.

Returns:

Zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired, or an error number to indicate the error.

Errors:**EAGAIN**

Couldn't acquire read lock because the maximum number of read locks for lock would be exceeded.

EDEADLK

The calling thread already holds a write lock on *rwlock*.

EINVAL

The value specified by *rwlock* doesn't refer to an initialized read-write lock object, or the *abs_timeout* nanosecond value is less than zero or greater than or equal to 1,000 million.

ETIMEDOUT

The lock couldn't be acquired before the specified timeout expired.

Classification:

[POSIX 1003.1 THR TMO](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_timedwrlock()

Lock a read-write lock for writing

Synopsis:

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(
    pthread_rwlock_t * rwlock,
    const struct timespec * abs_timeout );
```

Arguments:

rwlock

The read-write lock that you want to lock.

abs_timeout

A pointer to a [timespec](#) (p. 3473) that specifies the maximum time to wait to acquire the lock, expressed as an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlock_timedwrlock()` function applies a write lock to the read-write lock referenced by *rwlock* as in [pthread_rwlock_wrlock\(\)](#) (p. 2381).

However, if the lock can't be acquired without waiting for other threads to unlock the lock, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (i.e. when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The timeout is based on the `CLOCK_REALTIME` clock.

If the read-write lock can be locked immediately, the validity of the *abs_timeout* parameter isn't checked, and the function won't fail with a timeout.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to `pthread_rwlock_timedwrlock()`, upon return from the signal handler the thread resumes waiting for the lock as if it hadn't been interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*. The results are undefined if this function is called with an uninitialized read-write lock.

Returns:

Zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired, or an error number to indicate the error.

Errors:**EAGAIN**

Couldn't acquire read lock because the maximum number of read locks for lock would be exceeded.

EDEADLK

The calling thread already holds the *rwlock*.

EINVAL

The value specified by *rwlock* doesn't refer to an initialized read-write lock object, or the *abs_timeout* nanosecond value is less than zero or greater than or equal to 1,000 million.

ETIMEDOUT

The lock couldn't be acquired before the specified timeout expired.

Classification:

[POSIX 1003.1 THR TMO](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_tryrdlock()

Attempt to acquire a shared lock on a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(
    pthread_rwlock_t* rwlock );
```

Arguments:

rwlock

A pointer to a `pthread_rwlock_t` object that you want to lock for reading.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_rwlock_tryrdlock()* function attempts to acquire a shared lock on the read-write lock referenced by *rwlock*. If the read-write lock is already exclusively locked by any thread (including the calling thread), the function returns immediately instead of blocking until a read lock can be obtained.

Returns:

EOK

Success.

EAGAIN

On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.

EBUSY

The read-write lock was already write locked.

EDEADLK

The calling thread already has an exclusive lock for *rwlock*.

EFAULT

A fault occurred when the kernel tried to access *rw/*.

EINVAL

The read-write lock *rw/* is invalid.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_trywrlock()

Attempt to acquire an exclusive lock on a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_trywrlock(
    pthread_rwlock_t* rwl );
```

Arguments:

rwl

A pointer to a pthread_rwlock_t object that you want to lock for writing.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_rwlock_trywrlock()* function attempts to acquire an exclusive lock on the read-write lock referenced by *rwl*. If the read-write lock is already exclusively locked or shared locked, the function returns immediately instead of blocking until an exclusive lock can be obtained.

The function may need to block to determine the state of the read-write lock.

Returns:

EOK

Success.

EAGAIN

On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.

EBUSY

The read-write lock was already write locked or read locked.

EDEADLK

The calling thread already has an exclusive lock for *rw/*.

EFAULT

A fault occurred when the kernel tried to access *rw/*.

EINVAL

The read-write lock *rw/* is invalid.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_unlock()

Unlock a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_unlock( pthread_rwlock_t* rwlock );
```

Arguments:

rwlock

A pointer to a pthread_rwlock_t object that you want to unlock.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_rwlock_unlock()* function unlocks a read-write lock referenced by *rwlock*. The read-write lock may become available for any threads that were blocked on the read-write lock, depending on whether the read-write lock had been locked in exclusive or shared mode.



The read-write lock should be owned by the calling thread. If the calling thread doesn't hold the lock, no error status is returned, and the behavior of this read-write lock is now undefined.

Returns:

EOK

Success.

EAGAIN

On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.

EFAULT

A fault occurred when the kernel tried to access *rw/*.

EINVAL

The read-write lock *rw/* is invalid.

EPERM

No thread has a read or write lock on *rw/* or the calling thread doesn't have a write lock on *rw/*.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlock_wrlock()

Acquire an exclusive write lock on a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_wrlock(
    pthread_rwlock_t* rwl );
```

Arguments:

rwl

A pointer to a pthread_rwlock_t object that you want to lock for writing.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_rwlock_wrlock()* function acquires an exclusive lock on the read-write lock referenced by *rwl*. If the read-write lock is already shared-locked by any thread (including the calling thread) or exclusively-locked by any thread (other than the calling thread), the calling thread blocks until all shared locks and exclusive locks are released.

If a signal is delivered to a thread waiting to lock a read-write lock, it resumes waiting for the lock after returning from the signal handler.

Returns:

EOK

Success.

EAGAIN

On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.

EDEADLK

The calling thread already has an exclusive lock for *rwl*.

EFAULT

A fault occurred when the kernel tried to access *rw/*.

EINVAL

The read-write lock *rw/* is invalid.

Classification:

POSIX 1003.1 THR (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlockattr_destroy()

Destroy a read-write lock attribute object

Synopsis:

```
#include <pthread.h>

int pthread_rwlockattr_destroy(
    pthread_rwlockattr_t* attr );
```

Arguments:

attr

A pointer to the `pthread_rwlockattr_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlockattr_destroy()` function destroys a read-write lock attribute object created by [pthread_rwlockattr_init\(\)](#) (p. 2387).



Don't use a destroyed read-write lock attribute object reinitializing it by calling `pthread_rwlockattr_init()`.

Returns:

EOK

Success.

EINVAL

The object specified by *attr* is invalid.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlockattr_getpshared()

Get the process-shared attribute of a read-write lock attribute object

Synopsis:

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(
    const pthread_rwlockattr_t* attr,
    int* pshared );
```

Arguments:

attr

A pointer to the `pthread_rwlockattr_t` object that you want to get the attribute from.

pshared

A pointer to a location where the function can store the process-shared attribute.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlockattr_getpshared()` function gets the the process-shared attribute for the read-write lock attribute object specified by `attr`, storing it in `pshared`.

To let any thread with access to the read-write lock object's memory operate it, the process-shared attribute must be set to `PTHREAD_PROCESS_SHARED`, even if those threads are in different processes. Set the process-shared attribute to `PTHREAD_PROCESS_PRIVATE` to limit access to threads in the current process.

Returns:

EOK

Success.

EINVAL

The read-write lock attribute object specified by *attr* is invalid.

Classification:

POSIX 1003.1 THR TSH (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlockattr_init()

Create a read-write lock attribute object

Synopsis:

```
#include <pthread.h>

int pthread_rwlockattr_init(
    pthread_rwlockattr_t* attr );
```

Arguments:

attr

A pointer to the `pthread_rwlockattr_t` object that you want to initialize.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlockattr_init()` function initializes the specified read-write lock attribute object to its default values.

Changes made to a read-write lock attribute object changes after it's been used to initialize a read-write lock won't affect the previously initialized read-write locks.

Returns:

EOK

Success.

ENOMEM

There isn't enough memory available to initialize *attr*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_rwlockattr_setpshared()

Set the process-shared attribute of a read-write lock attribute object

Synopsis:

```
#include <pthread.h>

int pthread_rwlockattr_setpshared(
    pthread_rwlockattr_t* attr,
    int pshared );
```

Arguments:

attr

A pointer to the `pthread_rwlockattr_t` object that you want to set the attribute for.

pshared

The new value of the process-shared attribute; one of:

- `PTHREAD_PROCESS_SHARED` — let any thread with access to the read-write lock object's memory operate it, even if those threads are in different processes.
- `PTHREAD_PROCESS_PRIVATE` — limit access to threads in the current process.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlockattr_setpshared()` function sets the process-shared attribute for the read-write lock attribute object specified by *attr* to *pshared*.

Returns:

EOK

Success.

EINVAL

The *pshared* argument is invalid.

Classification:

POSIX 1003.1 THR TSH (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_self()

Get the calling thread's ID

Synopsis:

```
#include <pthread.h>

pthread_t pthread_self( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_self()` function returns the thread ID of the calling thread.

Returns:

The ID of the calling thread.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_setcancelstate()

Set a thread's cancellation state

Synopsis:

```
#include <pthread.h>

int pthread_setcancelstate( int state,
                           int* oldstate );
```

Arguments:

state

The new cancellation state.

oldstate

A pointer to a location where the function can store the old cancellation state.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_setcancelstate()* function sets the calling thread's cancellation state to *state* and returns the previous cancellation state in *oldstate*.

The cancellation state can have the following values:

PTHREAD_CANCEL_DISABLE

Cancellation requests are held pending.

PTHREAD_CANCEL_ENABLE

Cancellation requests may be acted on according to the cancellation type; see [pthread_setcanceltype\(\)](#) (p. 2394).

The default cancellation state for a thread is `PTHREAD_CANCEL_ENABLE`.



You can set this attribute (in a non-POSIX way) before creating the thread; for more information, see “[QNX Neutrino extensions](#) (p. 2285),” in the documentation for `pthread_create()`.

Returns:**EOK**

Success.

EINVAL

The cancellation state specified by *state* is invalid.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_setcanceltype()

Set a thread's cancellation type

Synopsis:

```
#include <pthread.h>

int pthread_setcanceltype( int type,
                          int* oldtype );
```

Arguments:

type

The new cancellation type.

oldtype

A pointer to a location where the function can store the old cancellation type.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_setcanceltype()* function sets the calling thread's cancellation type to *type* and returns the previous cancellation type in *oldtype*.

The cancellation type can have the following values:

PTHREAD_CANCEL_ASYNCHRONOUS

If cancellation is enabled, new or pending cancellation requests may be acted on immediately.

PTHREAD_CANCEL_DEFERRED

If cancellation is enabled, cancellation requests are held pending until a cancellation point.

The default cancellation state for a thread is `PTHREAD_CANCEL_DEFERRED`. Note that the standard POSIX and C library calls aren't asynchronous-cancellation safe.



You can set this attribute (in a non-POSIX way) before creating the thread; for more information, see “[QNX extensions](#) (p. 2285),” in the documentation for `pthread_create()`.

Returns:**EOK**

Success.

EINVAL

Invalid cancelability type *type*.

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_setconcurrency()

Set the concurrency level for a thread

Synopsis:

```
#include <pthread.h>

int pthread_setconcurrency( int new_level );
```

Arguments:

new_level

The new value for the concurrency level.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

QNX Neutrino doesn't support the multiplexing of user threads on top of several kernel scheduled entities. As such, the *pthread_setconcurrency()* and [pthread_getconcurrency\(\)](#) (p. 2294) functions are provided for source code compatibility but they have no effect when called. To maintain the function semantics, the *new_level* parameter is saved when *pthread_setconcurrency()* is called so that a subsequent call to *pthread_getconcurrency()* returns the same value.

Returns:

EOK

Success.

EINVAL

Negative argument *new_level*.

EAGAIN

The value specified by *new_level* would cause a system resource to be exceeded.

Classification:*POSIX 1003.1 XSI* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_setname_np()

Name a thread

Synopsis:

```
#include <pthread.h>

int pthread_setname_np(pthread_t tid,
                       const char* newname);
```

Arguments:

tid

The ID of the thread you want to name, or 0 if you want to name the calling thread.

newname

NULL, or a NULL-terminated string that specifies the new name. The maximum length is `_NTO_THREAD_NAME_MAX` (defined in `<sys/neutrino.h>`).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_setname_np()` function sets the name of the specified thread to *newname*. If *newname* is NULL, the function deletes any name already assigned to the thread. The “np” in the function's name stands for “non-POSIX.”



This function was added in the QNX Neutrino Core OS 6.3.2.

This function is currently implemented as follows:

- If a thread is setting its own name, `pthread_setname_np()` uses [ThreadCtl\(\)](#) (p. 3408).

- If a thread is setting another thread's name, *pthread_setname_np()* needs read/write access to the */proc/pid/as* entry for the process.

Only one program can have write access to the a process's entry in the */proc* filesystem at a time, so if another program (such as a debugger) already has write access to it, *pthread_setname_np()* fails with an error of **EBUSY**. For this reason, it's better to have a thread set its own name than have it set by another thread.

Returns:

EOK

Success.

E2BIG

The name is too long.

EBUSY

As described above, you're trying to name a thread other than the calling thread, and another program already has write access to */proc/pid/as*.

EPERM

You don't have the appropriate permissions to set the name.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_setschedparam()

Set thread scheduling parameters

Synopsis:

```
#include <pthread.h>

int pthread_setschedparam(
    pthread_t thread,
    int policy,
    const struct sched_param *param );
```

Arguments:

thread

The ID of the thread that you want to get the scheduling parameters for. You can get a thread ID by calling [pthread_create\(\)](#) (p. 2284) or [pthread_self\(\)](#) (p. 2391).

policy

The new scheduling policy; one of:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher priority thread.
- **SCHED_RR** — the same as **SCHED_FIFO**, except threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- **SCHED_OTHER** — currently the same as **SCHED_RR**.
- **SCHED_SPORADIC** — sporadic scheduling.

For more information, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

param

A pointer to a [sched_param](#) (p. 2731) structure that specifies the scheduling parameters that you want to use.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_setschedparam()` function sets the scheduling policy and associated scheduling parameters of thread *thread* to the values specified in *policy* and *param*.



In order to set the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROCMGR_AID_SCHEDULE` ability enabled. In order to change its priority to a value above the maximum permitted for unprivileged processes, your process must have the `PROCMGR_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

EOK

Success.

EINVAL

Invalid scheduling policy *policy* or parameters *param*.

ENOTSUP

Unsupported scheduling policy *policy* or parameters *param*.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

Invalid thread ID *thread*.

Classification:

[POSIX 1003.1 THR TPS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_setschedprio()

Set a thread's priority

Synopsis:

```
#include <pthread.h>

int pthread_setschedprio( pthread_t thread,
                          int prio );
```

Arguments:

thread

The ID of the thread that you want to get the scheduling parameters for. You can get a thread ID by calling [*pthread_create\(\)*](#) (p. 2284) or [*pthread_self\(\)*](#) (p. 2391).

prio

The new scheduling priority.

As an extension to POSIX in QNX Neutrino 6.6 or later, you can wrap the new priority in one these macros to specify how to handle out-of-range priority requests:

- *SCHED_PRIO_LIMIT_ERROR(prio)* — indicate an error
- *SCHED_PRIO_LIMIT_SATURATE(prio)* — saturate at the maximum allowed priority

If *procnto* was started with a *-P* option ending with *s* or *S*, out-of-range priority requests by default saturate at the maximum allowed value.

Library:

libc

Use the *-l c* option to *qcc* to link against this library. This library is usually included automatically.

Description:

The *pthread_setschedprio()* function sets the priority of thread *thread* to *prio*.

If the thread is running or runnable, the effect on its position in the ready queue depends on the direction of the modification:

- If you raise the priority, the thread becomes the tail of the ready queue for that priority.
- If you don't change the priority, the thread doesn't change position in the ready queue.
- If you lower the priority, the thread becomes the head of the ready queue for that priority.

The `pthread_setschedprio()` function gives you a way to temporarily raise a thread's priority and lower it again, without yielding to other threads of the same priority. This is necessary if you have to implement your own strategies for bounding priority inversion, such as priority inheritance or priority ceilings.



In order to change its priority to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

EOK

Success.

EINVAL

The priority isn't valid for the scheduling policy of the specified thread.

ENOTSUP

The specified priority isn't supported.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

Invalid thread ID *thread*.

Classification:

[POSIX 1003.1 THR TPS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

pthread_setspecific()

Set a thread-specific data value

Synopsis:

```
#include <pthread.h>

int pthread_setspecific( pthread_key_t key,
                        const void* value );
```

Arguments:

key

The key associated with the data that you want to set. See [pthread_key_create\(\)](#) (p. 2305).

value

The value that you want to store.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_setspecific()* function binds the thread specific data value *value* with the thread specific data key *key*.

You can call this function from within a thread-specific data destructor function.



You must call this function with a key that you got from *pthread_key_create()*. You can't use a key after destroying it with [pthread_key_delete\(\)](#) (p. 2308).

Returns:

EOK

Success.

ENOMEM

Insufficient memory to store thread specific data value *value*.

EINVAL

Invalid thread specific data key *key*.

Examples:

See [*pthread_key_create\(\)*](#) (p. 2305).

Classification:

[*POSIX 1003.1 THR*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Calling *pthread_setspecific()* with a non-NULL *value* may result in lost storage or infinite loops unless *value* was returned by *pthread_key_create()*.

pthread_sigmask()

Examine and change blocked signals

Synopsis:

```
#include <signal.h>

int pthread_sigmask( int how,
                    const sigset_t* set,
                    sigset_t* oset );
```

Arguments:

how

How you want to change the signal mask; one of:

- **SIG_BLOCK** — make the resulting signal mask the union of the current signal mask and the signal set *set*.
- **SIG_UNBLOCK** — make the resulting signal mask the intersection of the current signal mask and the complement of the signal set *set*.
- **SIG_SETMASK** — make the resulting signal mask the signal set *set*.

This argument is valid only if *set* is non-NULL.

set

A pointer to a `sigset_t` object that specifies the signals that you want to affect in the mask.

oset

NULL, or a pointer to a `sigset_t` object where the function can store the thread's old signal mask.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_sigmask()` function is used to examine and/or change the calling thread's signal mask. If *set* is non-NULL, the thread's signal mask is set to *set*. If *oset* is non-NULL, the thread's old signal mask is returned in *oset*.



In order to set the signal-blocked mask for a process with a different real or effective user ID, your process must have the `PROC_MGR_AID_SIGNAL` ability enabled. For more information, see [proc_mgr_ability\(\)](#) (p. 2141).

You can't block the `SIGKILL` and `SIGSTOP` signals.

Returns:

EOK

Success.

EINVAL

Invalid *how* parameter.

EPERM

The calling process doesn't have the required permission; see [proc_mgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 THR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_sleepon_broadcast()

Unblock waiting threads

Synopsis:

```
#include <pthread.h>

int pthread_sleepon_broadcast(
    const volatile void * addr );
```

Arguments:

addr

The handle that the threads are waiting on. The value of *addr* is typically a data structure that controls a resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_sleepon_broadcast()* function unblocks all threads currently waiting on *addr*. The threads are unblocked in priority order.

You should use *pthread_sleepon_broadcast()* or [pthread_sleepon_signal\(\)](#) (p. 2413), depending on the task you're doing:

Mapping a single predicate to one address

Use *pthread_sleepon_signal()*.

If you use *pthread_sleepon_broadcast()*, you must recheck the predicate and reblock if necessary. The first thread to wake up owns the lock; all others must go back to sleep.

If you use *pthread_sleepon_signal()*, you don't have to recheck the predicate. One thread owns the lock at a time.

Mapping multiple predicates to one address

You need to use *pthread_sleepon_broadcast()* to wake up all blocked threads. You must recheck the predicates and reblock if necessary. You should try to map only one predicate to one address.

Don't use *pthread_sleepon_signal()* in this case; it could result in a deadlock.

Returns:**EOK**

Success.

EINVAL

Invalid sleepon address.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_sleepon_lock()

Lock the pthread_sleepon functions*

Synopsis:

```
#include <pthread.h>

int pthread_sleepon_lock( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_sleepon_lock()* function calls [pthread_mutex_lock\(\)](#) (p. 2318) on a mutex associated with the *pthread_sleepon** class of functions. You should call this function before testing conditions that determine whether you need to call [pthread_sleepon_wait\(\)](#) (p. 2419), [pthread_sleepon_signal\(\)](#) (p. 2413), or [pthread_sleepon_broadcast\(\)](#) (p. 2409). This mutex prevents other threads from changing the conditions between the time you examine and act upon them.

This function may be implemented as a simple macro.

Returns:

EOK

Success.

EDEADLK

The calling thread already owns the controlling mutex.

EAGAIN

On the first use of *pthread_sleepon_lock()*, all kernel mutex objects were in use.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_sleepon_signal()

Signal a sleeping thread

Synopsis:

```
#include <pthread.h>

int pthread_sleepon_signal(
    const volatile void * addr );
```

Arguments:

addr

The handle that the threads are waiting on. The value of *addr* is typically a data structure that controls a resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_sleepon_signal()* function unblocks the highest priority thread waiting on *addr*.

You should use [pthread_sleepon_broadcast\(\)](#) (p. 2409) or *pthread_sleepon_signal()*, depending on the task you're doing:

Mapping a single predicate to one address

Use *pthread_sleepon_signal()*.

If you use *pthread_sleepon_broadcast()*, you must recheck the predicate and reblock if necessary. The first thread to wake up owns the lock; all others must go back to sleep.

If you use *pthread_sleepon_signal()*, you don't have to recheck the predicate. One thread owns the lock at a time.

Mapping multiple predicates to one address

You need to use *pthread_sleepon_broadcast()* to wake up all blocked threads. You must recheck the predicates and reblock if necessary. You should try to map only one predicate to one address.

Don't use *pthread_sleepon_signal()* in this case; it could result in a deadlock.

Returns:**EOK**

Success.

EINVAL

Invalid sleepon address.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_sleepon_timedwait()

Make a thread sleep while waiting

Synopsis:

```
#include <pthread.h>

int pthread_sleepon_timedwait(
    const volatile void * addr,
    uint64_t nsec );
```

Arguments:

addr

The handle that you want the thread to wait for. The value of *addr* is typically a data structure that controls a resource.

nsec

A limit on the amount of time to wait, in nanoseconds.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_sleepon_timedwait()* function uses a mutex and a condition variable to sleep on a handle, *addr*.

If *nsec* is nonzero, then *pthread_sleepon_timedwait()* calls [pthread_cond_timedwait\(\)](#) (p. 2265). If the *pthread_cond_timedwait()* times out, then *pthread_sleepon_timedwait()* returns ETIMEDOUT. If *nsec* is zero, then *pthread_sleepon_timedwait()* calls [pthread_cond_wait\(\)](#) (p. 2269) instead.

The *pthread_sleepon*()* functions provide a simple, uniform way to wait on a variety of resources in a multithreaded application. For example, a multithreaded filesystem may wish to wait on such diverse things as a cache block, a file lock, an operation complete and many others. For example, to wait on a resource:

```
pthread_sleepon_lock();
while((ptr = cachelist->free) == NULL) {
    pthread_sleepon_timedwait(cachelist, 1000);
}
```

```

cachelist->free = ptr->free;

pthread_sleepon_unlock();

```

To start an operation and wait upon its completion:

```

/* Line up for access to the driver */
pthread_sleepon_lock();
if(driver->busy) {
    pthread_sleepon_timedwait(&driver->busy, 1000);
}

/* We now have exclusive use of the driver */
driver->busy = 1;
driver_start(driver); /* This should be relatively fast */

/* Wait for something to signal driver complete */
pthread_sleepon_timedwait(&driver->complete, 1000);
pthread_sleepon_unlock();

/* Get the status/data */
driver_complete(driver);

/* Release control of the driver and signal anyone waiting */
pthread_sleepon_lock();
driver->busy = 0;
pthread_sleepon_signal(&driver->busy);
pthread_sleepon_unlock();

```

The use of a `while` loop instead of an `if` handles the case where the wait on *addr* is woken up using `pthread_sleepon_broadcast()`.

You must call `pthread_sleepon_lock()` (p. 2411), which acquires the controlling mutex for the condition variable and ensures that another thread won't enter the critical section between the test, block and use of the resource. Since `pthread_sleepon_timedwait()` calls `pthread_cond_timedwait()`, it releases the controlling mutex when it blocks. It reacquires the mutex before waking up.

The wakeup is accomplished by another thread's calling `pthread_sleepon_signal()` (p. 2413), which wakes up a single thread, or `pthread_sleepon_broadcast()` (p. 2409), which wakes up all threads blocked on *addr*. Threads are woken up in priority order. If there's more than one thread with the same highest priority, the one that has been waiting the longest is woken first.

A single mutex and one condition variable for each unique address that's currently being blocked on are used. The total number of condition variables is therefore equal to the number of unique *addrs* that have a thread waiting on them. This also means that the maximum number of condition variables never exceeds the number of threads. To accomplish this, condition variables are dynamically created as needed and placed upon an internal freelist for reuse when not.

You might find the `pthread_sleepon_*`(*)* functions easier to use and understand than condition variables. They also resemble the traditional `sleepon()` and `wakeup()` functions found in Unix kernels. They can be implemented as follows:

```

int _sleepon(void *addr) {
    int ret;

    if((ret = pthread_sleepon_lock()) == EOK) {
        ret = pthread_sleepon_timedwait(addr, 1000);
        pthread_sleepon_unlock();
    }
}

```

```

    return ret;
}

void _wakeup(void *addr) {
    if(pthread_sleepon_lock() == EOK) {
        pthread_sleepon_broadcast(addr, 1000);
        pthread_sleepon_unlock();
    }
}

```

Note that in most Unix kernels, a thread runs until it blocks and thus need not worry about protecting the condition it checks with a mutex. Likewise when a Unix *wakeup()* is called, there isn't an immediate thread switch. Therefore, you can use only the above simple routines (*_wakeup()* and *_sleepon()*) if all your threads run with *SCHED_FIFO* scheduling and at the same priority, thus more closely mimicking Unix kernel scheduling.

Returns:

EDEADLK

The calling thread already owns the controlling mutex.

ETIMEDOUT

The time specified by *nsec* has passed.

EOK

Success.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_sleepon_unlock()

Unlock the pthread_sleepon() functions*

Synopsis:

```
#include <pthread.h>

int pthread_sleepon_unlock( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pthread_sleepon_unlock()* function calls [pthread_mutex_unlock\(\)](#) (p. 2329) on a mutex associated with the *pthread_sleepon*()* class of functions. You should call it at the end of a critical section entered by [pthread_sleepon_lock\(\)](#) (p. 2411).

This function may be implemented as a simple macro.

Returns:

- EOK**
Success.
- EPERM**
The current thread doesn't own the controlling mutex.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_sleepon_wait()

Make a thread sleep while waiting

Synopsis:

```
#include <pthread.h>

int pthread_sleepon_wait( const volatile void * addr );
```

Arguments:

addr

The handle that you want the thread to wait for. The value of *addr* is typically a data structure that controls a resource.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_sleepon_wait()* function uses a mutex and a condition variable to sleep on a handle, *addr*.

The *pthread_sleepon** functions provide a simple, uniform way to wait on a variety of resources in a multithreaded application. For example, a multithreaded filesystem may wish to wait on such diverse things as a cache block, a file lock, an operation complete and many others. For example, to wait on a resource:

```
pthread_sleepon_lock();
while((ptr = cachelist->free) == NULL) {
    pthread_sleepon_wait(cachelist);
}
cachelist->free = ptr->free;
pthread_sleepon_unlock();
```

To start an operation and wait for its completion:

```
/* Line up for access to the driver */
pthread_sleepon_lock();
if(driver->busy) {
    pthread_sleepon_wait(&driver->busy);
}

/* We now have exclusive use of the driver */
driver->busy = 1;
driver_start(driver); /* This should be relatively fast */
```

```

/* Wait for something to signal driver complete */
pthread_sleepon_wait(&driver->complete);
pthread_sleepon_unlock();

/* Get the status/data */
driver_complete(driver);

/* Release control of the driver and signal anyone waiting */
pthread_sleepon_lock();
driver->busy = 0;
pthread_sleepon_signal(&driver->busy);
pthread_sleepon_unlock();

pthread_exit(NULL);

```

Choose carefully when you decide whether to use a `while` loop

- If the wait on *addr* is woken up using `pthread_sleepon_broadcast()` (p. 2409), you must use a `while` loop.
- If threads are woken up using `pthread_sleepon_signal()` (p. 2413), you *may* use the `if` conditional if the design of the program guarantees proper synchronization and scheduling among contending threads. This is guaranteed in the above example, assuming that none of the threads attempt to reacquire the driver resource (i.e. `pthread_exit()` call).

If you're in doubt, use a `while` loop, because it guarantees access to the desired resource.

You must call `pthread_sleepon_lock()` (p. 2411), which acquires the controlling mutex for the condition variable and ensures that another thread won't enter the critical section between the test, block and use of the resource. Since `pthread_sleepon_wait()` calls `pthread_cond_wait()` (p. 2269), it releases the controlling mutex when it blocks. It reacquires the mutex before waking up.

The wakeup is accomplished by another thread's calling `pthread_sleepon_signal()` (p. 2413), which wakes up a single thread, or `pthread_sleepon_broadcast()` (p. 2409), which wakes up all threads blocked on *addr*. Threads are woken up in priority order. If there's more than one thread with the same highest priority, the one that has been waiting the longest is woken first.

A single mutex and one condition variable for each unique address that's currently being blocked on are used. The total number of condition variables is therefore equal to the number of unique *addrs* that have a thread waiting on them. This also means that the maximum number of condition variables never exceeds the number of threads. To accomplish this, condition variables are dynamically created as needed and placed upon an internal freelist for reuse when not.

You might find the `pthread_sleepon_*` functions easier to use and understand than condition variables. They also resemble the traditional `sleepon()` and `wakeup()` functions found in Unix kernels. They can be implemented as follows:

```

int _sleepon(void *addr) {
    int ret;

    if((ret = pthread_sleepon_lock()) == EOK) {
        ret = pthread_sleepon_wait(addr);
    }
}

```



```

        pthread_sleepon_unlock();
    }
    return ret;
}

void _wakeup(void *addr) {
    if(pthread_sleepon_lock() == EOK) {
        pthread_sleepon_broadcast(addr);
        pthread_sleepon_unlock();
    }
}

```

Note that in most Unix kernels, a thread runs until it blocks, and thus need not worry about protecting the condition it checks with a mutex. Likewise, when a Unix *wakeup()* is called, there isn't an immediate thread switch. Therefore, you can use only the above simple routines (*_wakeup()* and *_sleepon()*) if all your threads run with SCHED_FIFO scheduling and at the same priority, thus more closely mimicking Unix kernel scheduling.

Returns:

EOK

Success.

EDEADLK

The calling thread already owns the controlling mutex.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_spin_destroy()

Destroy a thread spinlock

Synopsis:

```
#include <pthread.h>

int pthread_spin_destroy(
    pthread_spinlock_t * spinner );
```

Arguments:

spinner

A pointer to the `pthread_spinlock_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_spin_destroy()` function destroys the thread spinlock *spinner*, releasing its resources.

Once you've destroyed the spinlock, don't use it again until you've reinitialized it by calling [pthread_spin_init\(\)](#) (p. 2424).

Calling `pthread_spin_destroy()` gives undefined results when a thread has *spinner* locked or when *spinner* isn't initialized.

Returns:

EOK

Success.

EBUSY

The thread spinlock *spinner* is in use by another thread and can't be destroyed.

EINVAL

Invalid `pthread_spinlock_t` object *spinner*.

Classification:*POSIX 1003.1 THR SPI* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_spin_init()

Initialize a thread spinlock

Synopsis:

```
#include <pthread.h>

int pthread_spin_init( pthread_spinlock_t * spinner,
                      int pshared );
```

Arguments:

spinner

A pointer to the `pthread_spinlock_t` object that you want to initialize.

pshared

The value that you want to use for the process-shared attribute of the spinlock. The possible values are:

- `PTHREAD_PROCESS_SHARED` — the spinlock may be operated on by any thread that has access to the memory where the spinlock is allocated, even if it's allocated in memory that's shared by multiple processes.
- `PTHREAD_PROCESS_PRIVATE` — the spinlock can be operated on only by threads created within the same process as the thread that initialized the spinlock. If threads of differing processes attempt to operate on such a spinlock, the behavior is undefined.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_spin_init()` function allocates the resources required for the thread spinlock *spinner*, and initializes *spinner* to an unlocked state.

Any thread that can access the memory where *spinner* is allocated can operate on the spinlock.

Results are undefined if you call `pthread_spin_init()` on a *spinner* that's already initialized, or if you try to use a spinlock that hasn't been initialized.

Returns:

Zero on success, or an error number to indicate the error.

Errors:**EAGAIN**

The system doesn't have the resources required to initialize a new spinlock.

EBUSY

The process spinlock, *spinner*, is in use by another thread and can't be initialized.

EINVAL

Invalid pthread_spinlock_t object *spinner*.

ENOMEM

The system doesn't have enough free memory to create the new spinlock.

Classification:

[POSIX 1003.1 THR SPI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_spin_lock()

Lock a thread spinlock

Synopsis:

```
#include <pthread.h>

int pthread_spin_lock( pthread_spinlock_t * spinner );
```

Arguments:

spinner

A pointer to the `pthread_spinlock_t` object that you want to lock.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_spin_lock()` function locks the thread spinlock specified by *spinner*. If *spinner* isn't immediately available, `pthread_spin_lock()` blocks until *spinner* can be locked.

If a thread attempts to lock a spinlock that's already locked via `pthread_spin_lock()` or [pthread_spin_trylock\(\)](#) (p. 2428), the thread returns EDEADLK.

Returns:

EOK

Success.

EAGAIN

Insufficient resources available to lock *spinner*.

EDEADLK

The calling thread already holds *spinners* lock.

EINVAL

Invalid `pthread_spinlock_t` object *spinner*.

Classification:

POSIX 1003.1 THR SPI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

You may not get the desired behavior from this function because the current implementation is through mutexes. In the uncontested case, this gives the same behavior as spinlocks. In the contested case, this function makes a kernel call.

pthread_spin_trylock()

Try to lock a thread spinlock

Synopsis:

```
#include <pthread.h>

int pthread_spin_trylock(
    pthread_spinlock_t * spinner );
```

Arguments:

spinner

A pointer to the `pthread_spinlock_t` object that you want to try to lock.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_spin_trylock()* function attempts to lock the thread spinlock specified by *spinner*. It returns immediately if *spinner* can't be locked.

If a thread attempts to lock a spinlock that it's already locked via [pthread_spin_lock\(\)](#) (p. 2426) or *pthread_spin_trylock()*, the thread deadlocks.

Returns:

EOK

Success.

EAGAIN

Insufficient resources available to lock *spinner*.

EBUSY

The thread spinlock *spinner* is already locked by another thread.

EINVAL

Invalid `pthread_spinlock_t` object *spinner*.

Classification:*POSIX 1003.1 THR SPI* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_spin_unlock()

Unlock a thread spinlock

Synopsis:

```
#include <pthread.h>

int pthread_spin_unlock( spinlock_t * spinner );
```

Arguments:

spinner

A pointer to the `pthread_spinlock_t` object that you want to unlock.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_spin_unlock()` function unlocks the thread spinlock specified by *spinner*, which was locked with [pthread_spin_lock\(\)](#) (p. 2426) or [pthread_spin_trylock\(\)](#) (p. 2428).

If there are threads spinning on *spinner*, the spinlock becomes available, and an unspecified thread acquires the lock.

Returns:

EOK

Success.

EINVAL

Invalid process spinlock *spinner*.

EPERM

The calling thread doesn't hold the lock.

Classification:

[POSIX 1003.1 THR SPI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_testcancel()

Test thread cancellation

Synopsis:

```
#include <pthread.h>

void pthread_testcancel( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_testcancel()* function creates a cancellation point in the calling thread. This function has no effect if cancellation is disabled.

Classification:

[*POSIX 1003.1 THR*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

pthread_timedjoin(), pthread_timedjoin_monotonic()

Join a thread, with a time limit

Synopsis:

```
#include <pthread.h>

int pthread_timedjoin(
    pthread_t thread,
    void** value_ptr,
    const struct timespec* abstime );

int pthread_timedjoin_monotonic(
    pthread_t thread,
    void** value_ptr,
    const struct timespec* abstime );
```

Arguments:

thread

The target thread whose termination you're waiting for.

value_ptr

NULL, or a pointer to a location where the function can store the value passed to [pthread_exit\(\)](#) (p. 2292) by the target thread.

abstime

A pointer to a [timespec](#) (p. 3473) structure that specifies the maximum time to wait for the join, expressed as an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *pthread_timedjoin()* and *pthread_timedjoin_monotonic()* functions are QNX Neutrino functions that are similar to the POSIX [pthread_join\(\)](#) (p. 2303) function, except that they give an error of ETIMEDOUT if the join doesn't occur before the absolute time specified by *abstime* passes (i.e. the system time is greater than or equal to *abstime*):

If you are not too long, I will wait here for you all my life.

— Oscar Wilde, *The Importance of Being Earnest*

The `pthread_timedjoin_monotonic()` function uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

The `pthread_timedjoin()` and `pthread_timedjoin_monotonic()` functions block the calling thread until the target thread *thread* terminates, unless *thread* has already terminated. If *value_ptr* is non-NULL and the functions return successfully, then the value passed to `pthread_exit()` by the target thread is placed in *value_ptr*. If the target thread has been canceled, then *value_ptr* is set to `PTHREAD_CANCELED`.

The target thread must be joinable. Multiple `pthread_join()`, `pthread_timedjoin()`, `pthread_timedjoin_monotonic()`, `ThreadJoin()`, and `ThreadJoin_r()` calls on the same target thread aren't allowed. When `pthread_timedjoin()` or `pthread_timedjoin_monotonic()` returns successfully, the target thread has been terminated.

Returns:

EOK

Success.

EBUSY

The thread *thread* is already being joined.

EDEADLK

The thread *thread* is the calling thread.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

The thread *thread* isn't joinable.

ESRCH

The thread *thread* doesn't exist.

ETIMEDOUT

The absolute time specified in *abstime* passed before the join occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ptsname()

Get the name of a slave pseudo-terminal device

Synopsis:

```
#include <stdlib.h>

char *ptsname( int fildes );
```

Arguments:

fildes

The file descriptor for the *master* pseudo-terminal.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *ptsname()* function gets the name of the slave pseudo-terminal device associated with a master pseudo-terminal device.

The [*ptsname_r\(\)*](#) (p. 2438) function is a QNX Neutrino function that's a reentrant version of *ptsname()*.

Returns:

A pointer to a string containing the pathname of the corresponding slave device, or NULL if an error occurred (e.g. *fildes* is an invalid file descriptor, or the slave device name doesn't exist in the filesystem).

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	No

ptsname_r()

Get the name of a slave pseudo-terminal device

Synopsis:

```
#include <stdlib.h>

char *ptsname_r( int fildes,
                 char *buffer,
                 size_t buflen );
```

Arguments:

fildes

The file descriptor for the *master* pseudo-terminal.

buffer

A pointer to a buffer where the function can store the name.

buflen

The length of the buffer, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *ptsname_r()* function gets the name of the slave pseudo-terminal device associated with a master pseudo-terminal device. It's a reentrant version of [ptsname\(\)](#) (p. 2436).

Returns:

A pointer to a string containing the pathname of the corresponding slave device, or `NULL` if an error occurred (e.g. *fildes* is an invalid file descriptor, or the slave device name doesn't exist in the filesystem).

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

`_pulse`

Structure that describes a pulse

Synopsis:

```
#include <sys/neutrino.h>

struct _pulse {
    uint16_t          type;
    uint16_t          subtype;
    int8_t            code;
    uint8_t           zero[3];
    union sigval      value;
    int32_t           scoid;
};
```

Description:

The `_pulse` structure describes a *pulse*, a fixed-size, nonblocking message that carries a small payload (four bytes of data plus a single byte code). The members include:

type

`_PULSE_TYPE (0)`

subtype

`_PULSE_SUBTYPE (0)`

code

A code that identifies the type of pulse. The QNX Neutrino RTOS reserves the negative codes, including the following:

- `_PULSE_CODE_UNBLOCK`
- `_PULSE_CODE_DISCONNECT`
- `_PULSE_CODE_THREADDEATH`
- `_PULSE_CODE_COIDDEATH`
- `_PULSE_CODE_NET_ACK`, `_PULSE_CODE_NET_UNBLOCK`, and `_PULSE_CODE_NET_DETACH` — reserved for the `io-pkt*` resource manager.

You can define your own pulses, with a code in the range from `_PULSE_CODE_MINAVAIL` through `_PULSE_CODE_MAXAVAIL`.

value

Information that's relevant to the code:

- `_PULSE_CODE_UNBLOCK` — the receive ID (*rcvid*) associated with the blocking message.
- `_PULSE_CODE_DISCONNECT` — no value defined.
- `_PULSE_CODE_THREADDEATH` — the thread ID of the thread that died.
- `_PULSE_CODE_COIDDEATH` — the connection ID of a connection that was attached to a destroyed channel.

For more details, see [ChannelCreate\(\)](#) (p. 278).

If you define your own pulses, you can decide what information you want to store in this field.

soid

Server connection ID.

Classification:

[QNX Neutrino](#) (p. 46)

pulse_attach()

Attach a handler function to a pulse code

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int pulse_attach( dispatch_t * dpp,
                  int flags,
                  int code,
                  int (* func)
                    ( message_context_t * ctp,
                      int code,
                      unsigned flags,
                      void * handle ),
                  void * handle );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433).

flags

Currently, the following flag is defined in `<sys/dispatch.h>`:

- `MSG_FLAG_ALLOC_PULSE` — allocate and attach a pulse code that's different than any other code that was either given to *pulse_attach()* through the *code* argument, or allocated by *pulse_attach()*. The allocated code is in the range `_PULSE_CODE_MINAVAIL` and `_PULSE_CODE_MAXAVAIL`.

code

The pulse code that you want to attach the function to.

func

The function that you want to call when a message in the given range is received; see below, as well as “[Handler function](#) (p. 1626)” in the documentation for *message_attach()*.

handle

An arbitrary handle that you want to associate with data for the defined message range. This handle is passed to *func*.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pulse_attach()* function attaches a pulse *code* to a user-supplied function *func*. You can use the same function *func* with *message_attach()* (p. 1624). By examining *ctp->rcvid*, the *func* function can determine whether a pulse or message was received.

When the resource manager receives a pulse that matches *code*, it calls *func*. This user-supplied function is responsible for doing any specific work needed to handle the pulse pointed to by *ctp->msg.pulse*. The *handle* passed to the function is the *handle* initially passed to *pulse_attach()*. The *handle* may be a device entry you want associated with the pulse *code*. Your function should return 0; other return values are reserved.

You typically use *pulse_attach()* to associate pulses generated by interrupt handlers or timers with a routine in the main program of your resource manager.

Returns:

If `MSG_FLAG_ALLOC_PULSE` is specified, the function returns the allocated pulse code; otherwise, it returns the *code* that's passed in. On failure, -1 is returned (*errno* (p. 539) is set).

Errors:

EAGAIN

Couldn't allocate a pulse *code*.

EINVAL

The pulse *code* is out of range, or it's already registered.

ENOMEM

Insufficient memory to allocate internal data structures.

Examples:

```
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
```

```

#include <stdlib.h>

int my_func( ... ) {
    ...
}

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int flag = 0, code, mycode;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
            dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    ...

    mycode = ...;

    if ( (code = pulse_attach( dpp, flag, mycode,
        &my_func, NULL)) == -1 ) {
        fprintf ( stderr, "Failed to attach code %d.\n", mycode );
        return 1;
    }
    /* else successfully attached a pulse code */

    ...
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

pulse_detach()

Detach a handler function from a pulse code

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int pulse_detach( dispatch_t * dpp,
                  int code,
                  int flags );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433).

code

The pulse code that you want to detach.

flags

Reserved; pass 0 for this argument.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *pulse_detach()* function detaches the pulse *code*, for dispatch handle *dpp*, that was attached with [pulse_attach\(\)](#) (p. 2442).

Returns:

0

Success.

-1

The pulse *code* doesn't match any attached pulse code.

Examples:

```

#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int my_func( ... ) {
    ...
}

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int flag=0, code, mycode;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
            dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }

    ...
    if ( (code = pulse_attach( dpp, flag, mycode,
        &my_func, NULL)) == -1 ) {
        fprintf ( stderr, "Failed to attach pulse code %d.\n", \
            mycode );
        return 1;
    }

    ...

    if ( pulse_detach ( dpp, code, flag ) == -1 ) {
        fprintf ( stderr, "Failed to detach code %d.\n", code );
        return 1;
    }
    /* else message was detached */

    ...
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

putc()

Write a character to a stream

Synopsis:

```
#include <stdio.h>

int putc( int c,
          FILE* fp );
```

Arguments:

c

The character that you want to write.

fp

The stream you want to write the character on.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putc()` macro writes the character `c`, cast as `(int)(unsigned char)`, to the output stream designated by `fp`.

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurs ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE* fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF ) {
            putc( c, stdout );
        }
    }
}
```

```
        fclose( fp );  
    }  
    return EXIT_SUCCESS;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

putc() is a macro.

putc_unlocked()

Write a character to an output stream

Synopsis:

```
#include <stdio.h>

int putc_unlocked( int c,
                   FILE *stream );
```

Arguments:

c

The character that you want to write.

stream

The stream you want to write the character on.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *putc_unlocked()* function is a thread-unsafe version of [putc\(\)](#) (p. 2447). You can use it safely only when the invoking thread has locked *stream* using [flockfile\(\)](#) (p. 669) (or [ftrylockfile\(\)](#) (p. 796)) and [funlockfile\(\)](#) (p. 801).

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred ([errno](#) (p. 539) is set).

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

putchar()

Write a character to standard output

Synopsis:

```
#include <stdio.h>

int putchar( int c );
```

Arguments:

c

The character that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putchar()` function writes the character `c`, cast as `(int)(unsigned char)`, to the `stdout` stream. It's equivalent to:

```
fputc( c, stdout );
```

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurs ([*errno*](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    c = fgetc( fp );
    while( c != EOF ) {
        putchar( c );
        c = fgetc( fp );
    }
    fclose( fp );
}
```

```
    return EXIT_SUCCESS;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

putchar_unlocked()

Write a character to stdout

Synopsis:

```
#include <stdio.h>

int putchar_unlocked( int c );
```

Arguments:

c

The character that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putchar_unlocked()` function is a thread-unsafe version of [putchar\(\)](#) (p. 2451). You can use it safely only when the invoking thread has locked `stdout` using [flockfile\(\)](#) (p. 669) (or [ftrylockfile\(\)](#) (p. 796)) and [funlockfile\(\)](#) (p. 801).

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred ([errno](#) (p. 539) is set).

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

putenv()

Add, change or delete an environment variable

Synopsis:

```
#include <stdlib.h>

int putenv( char *env_name );
```

Arguments:

env_name

The name of the environment, and what you want to do to it; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *putenv()* function uses *env_name*, in the form *name=value*, to set the environment variable *name* to *value*. This function alters *name* if it exists, or creates a new environment variable.

In either case, *env_name* becomes part of the environment; subsequent modifications to the string pointed to by *env_name* affect the environment.

The space for environment names and their values is limited. Consequently, *putenv()* can fail when there's insufficient space remaining to store an additional value.



If *env_name* isn't a literal string, you should duplicate the string, since *putenv()* doesn't copy the value. For example:

```
putenv( strdup( buffer ) );
```

Returns:

0

Success.

-1

An error occurred; *errno* is set.

Errors:

ENOMEM

There wasn't enough memory to expand the environment.

Examples:

The following gets the string currently assigned to **INCLUDE** and displays it, assigns a new value to it, gets and displays it, and then removes **INCLUDE** from the environment.

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *path;
    path = getenv( "INCLUDE" );
    if( path != NULL ) {
        printf( "INCLUDE=%s\n", path );
    }

    if( putenv( "INCLUDE=/src/include" ) != 0 ) {
        printf( "putenv() failed setting INCLUDE\n" );
        return EXIT_FAILURE;
    }

    path = getenv( "INCLUDE" );
    if( path != NULL ) {
        printf( "INCLUDE=%s\n", path );
    }

    unsetenv( "INCLUDE" );

    return EXIT_SUCCESS;
}
```

This program produces the following output:

```
INCLUDE=/usr/nto/include
INCLUDE=/src/include
```

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

Never use *putenv()* with an automatic variable.

The *putenv()* function manipulates the environment pointed to by the global [environ](#) (p. 528) variable.

puts()

Write a string to stdout

Synopsis:

```
#include <stdio.h>

int puts( const char *buf );
```

Arguments:

buf

A pointer to the zero-terminated string that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `puts()` function writes the character string pointed to by *buf* to the *stdout* stream, and appends a newline character to the output. The terminating NUL character of *buf* isn't written.

Returns:

A nonnegative value for success, or EOF if an error occurs ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    char buffer[80];

    fp = freopen( "file", "r", stdin );
    while( gets( buffer ) != NULL ) {
        puts( buffer );
    }
    fclose( fp );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

putspent()

Put an entry into the shadow password database

Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

int putspent( const struct spwd* p,
              FILE* fp );
```

Arguments:

p

A pointer to a `spwd` structure that contains the entry that you want to write.

fp

The stream that you want to write the entry on.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putspent()` function writes a shadow password entry into the specified file. This function is the inverse of [getspent\(\)](#) (p. 971).

Given a pointer to a `spwd` structure created by the [getspent\(\)](#) (p. 971) or the [getsppnam\(\)](#) (p. 974) routine, `putspent()` writes a line on the stream `fp`, which matches the format of `</etc/shadow>`. The `spwd` structure contains the following members:

```
char    *sp_namp;      /* name */
char    *sp_pwdp;      /* encrypted password */
long    sp_lstchg;     /* last changed */
long    sp_max;        /* #days (min) to change */
long    sp_min;        /* #days (max) to change */
long    sp_warn;       /* #days to warn */
long    sp_inact;      /* #days of inactivity */
long    sp_expire;     /* date to auto-expire */
long    sp_flag;       /* reserved */
```

If the `sp_min`, `sp_max`, `sp_lstchg`, `sp_warn`, `sp_inact`, or `sp_expire` field of the structure is `-1`, or if `sp_flag = 0`, the corresponding `</etc/shadow>` field is cleared.

Returns:

Zero.

Errors:

The *putspent()* function uses the following functions, and as a result *errno* can be set to an error for any of these calls:

- [fclose\(\)](#) (p. 618)
- [fgets\(\)](#) (p. 652)
- [fopen\(\)](#) (p. 681)
- [fseek\(\)](#) (p. 772)
- [rewind\(\)](#) (p. 2646)

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * This program adds a user and password to
 * a temporary file which can then be used with
 * fgetspent() (of course the password
 * string should be encrypted already etc.)
 */

int main( int argc, char** argv )
{
    FILE* fp;
    struct spwd sp;
    char pwbuf[80], nambuf[80];

    memset(&sp, 0, sizeof(sp));
    if (argc < 2) {
        printf("%s filename \n", argv[0]);
        return(EXIT_FAILURE);
    }

    if (!(fp = fopen(argv[1], "w"))) {
        fprintf(stderr, "Can't open file %s \n", argv[1]);
        perror("Problem ");
        return(1);
    }

    printf("Enter a userid: ");
    if (!gets(nambuf)) {
        fprintf(stderr, "Can't get username string\n");
    }
    sp.sp_namp = nambuf;

    printf("Enter a password: ");
    if (!gets(pwbuf)) {
        fprintf(stderr, "Can't get username password\n");
    }
    sp.sp_pwdp = pwbuf;

    putspent(&sp, fp);
    fclose(fp);
    return( EXIT_SUCCESS );
}
```


Classification:*Unix* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

pututline()

Write an entry in the user-information file

Synopsis:

```
#include <utmp.h>

void pututline( struct utmp * __utmp );
```

Arguments:

__utmp

A pointer to the [utmp](#) (p. 3594) structure for the entry that you want to add.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *pututline()* function writes out the supplied `utmp` structure into the `utmp` file.

It uses [getutid\(\)](#) (p. 987) to search forward for the proper place if it finds that it isn't already there. Normally, you should search for the proper entry by calling [getutent\(\)](#) (p. 985), [getutid\(\)](#), or [getutline\(\)](#) (p. 989). If so, *pututline()* doesn't search. If *pututline()* doesn't find a matching slot for the new entry, it adds a new entry to the end of the file.

When called by a non-root user, *pututline()* invokes a *setuid()* root program to verify and write the entry, since the file specified in `_PATH_UTMP` is normally writable only by root. In this event, the *ut_name* field must correspond to the actual user name associated with the process; the *ut_type* field must be either `USER_PROCESS` or `DEAD_PROCESS`; the *ut_line* field must be a device-special file and be writable by the user.

Returns:

A pointer to the [utmp](#) (p. 3594) structure.

Files:**`_PATH_UTMP`**

Specifies the user information file.

Classification:[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either *getutid()* or *getutline()*, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it's searching for, the function looks no further. For this reason, to use *getutline()* to search for multiple occurrences, zero out the static area after each success, or *getutline()* will return the same structure over and over again.

There's one exception to the rule about emptying the structure before further reads are done: the implicit read done by *pututline()* (if it finds that it isn't already at the correct place in the file) doesn't hurt the contents of the static structure returned by the *getutent()*, *getutid()* or *getutline()* routines, if the user has just modified those contents and passed the pointer back to *pututline()*.

These routines use buffered standard I/O for input, but *pututline()* uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

putw()

Put a word on a stream

Synopsis:

```
#include <wchar.h>

int putw( int w,
          FILE *stream );
```

Arguments:

w

The word that you want to write.

stream

The stream that you want to write a word on.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `putw()` function writes the C `int` (word) `w` to the standard I/O output `stream` (at the position of the file pointer, if defined). The size of a word is the size of an integer, and varies from machine to machine. The `putw()` function neither assumes nor causes special alignment in the file.

Returns:

0

Success.

1

An error occurred; `errno` is set.

Errors:**EFBIG**

The file is a regular file and an attempt was made to write at or beyond the offset maximum.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Because of possible differences in word length and byte ordering, files written using *putw()* are machine-dependent, and might not be read correctly using *getw()* on a different processor.

putwc()

Write a wide character to a stream

Synopsis:

```
#include <wchar.h>

wint_t putwc( wchar_t wc,
              FILE * fp );
```

Arguments:

wc

The wide character that you want to write.

fp

The stream that you want to write the wide character on.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *putwc()* functions writes the wide character specified by *wc*, cast as `(wint_t)(wchar_t)`, to the output stream specified by *fp*.

Returns:

The wide character written, cast as `(wint_t)(wchar_t)`, or `WEOF` if an error occurs ([errno](#) (p. 539) is set).



If *wc* exceeds the valid wide-character range, the value returned is the wide character written, not *wc*.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *fp* and would have been blocked by this operation.

EBADF

The file descriptor for *fp* isn't valid for writing.

EFBIG

The file exceeds the maximum file size, the process's file size limit, or the function can't write at or beyond the offset maximum.

EINTR

A signal terminated the write operation; no data was transferred.

EIO

A physical I/O error has occurred or all of the following conditions were met:

- The process is in the background.
- TOSTOP is set.
- The process is blocking/ignoring SIGTTOU.
- The process group is orphaned.

EPIPE

Can't write to pipe or FIFO because it's closed; a SIGPIPE signal is also sent to the thread.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

putwchar()

Write a wide character to a stdout

Synopsis:

```
#include <wchar.h>

wint_t putwchar( wchar_t wc );
```

Arguments:

wc

The wide character that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *putwchar()* function writes the wide character *wc*, cast as `(wint_t)(wchar_t)`, to the *stdout* stream. It's equivalent to:

```
fputwc( wc, stdout );
```

Returns:

The wide character written, cast as `(wint_t)(wchar_t)` or `WEOF` if an error occurs (*errno* (p. 539) is set).



If *wc* exceeds the valid wide-character range, the value returned is the wide character written, not *wc*.

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for *fp* and would have been blocked by this operation.

EBADF

The file descriptor for *fp* isn't valid for writing.

EFBIG

The file exceeds the maximum file size, the process's file size limit, or the function can't write at or beyond the offset maximum.

EINTR

A signal terminated the write operation; no data was transferred.

EIO

A physical I/O error has occurred or all of the following conditions were met:

- The process is in the background.
- TOSTOP is set.
- The process is blocking/ignoring SIGTTOU.
- The process group is orphaned.

EPIPE

Can't write to pipe or FIFO because it's closed; a SIGPIPE signal is also sent to the thread.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

pwrite(), pwrite64()

Write into a file without changing the file pointer

Synopsis:

```
#include <unistd.h>

ssize_t pwrite( int filedes,
               const void* buff,
               size_t nbytes,
               off_t offset );

ssize_t pwrite64( int filedes,
                 const void* buff,
                 size_t nbytes,
                 off64_t offset );
```

Arguments:

filedes

The file descriptor for the file you want to write in.

buff

A pointer to a buffer that contains the data you want to write.

nbytes

The number of bytes to write.

offset

The desired position inside the file.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *pwrite()* and *pwrite64()* functions perform the same action as [write\(\)](#) (p. 3737), except that they write into a given position without changing the file pointer. The *pwrite64()* function is a 64-bit version of *pwrite()*.

Returns:

The number of bytes actually written, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor, *fdes*, isn't a valid file descriptor open for writing.

EFBIG

One of the following occurred:

- An attempt was made to write a file that exceeds the maximum file size or the process's file size limit, and there was no room for any bytes to be written.
- The file is a regular file, *nbytes* is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fdes*.

EINTR

The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EINVAL

The offset argument is invalid; the value is negative. The file pointer remains unchanged.

EIO

One of the following occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU`, and the process group of the process is orphaned.
- A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOBUFS

Insufficient resources were available in the system to perform the operation.

ENOSPC

There's no free space remaining on the device containing the file.

ENOSYS

The *pwrite()* function isn't implemented for the filesystem specified by *fildes*.

ENXIO

One of the following occurred:

- A request was made of a nonexistent device, or the request was outside the capabilities of the device.
- A hangup occurred on the STREAM being written to.

EPIPE

An attempt was made to write to a pipe (or FIFO) that isn't open for reading by any process. A SIGPIPE signal is also sent to the process.

ERANGE

The transfer request size was outside the range supported by the STREAMS file associated with *fildes*.

ESPIPE

The *fildes* is associated with a pipe or FIFO. The file pointer remains unchanged.

Classification:

pwrite() is [POSIX 1003.1 XSI](#) (p. 46); *pwrite64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Chapter 19

Q

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “Q”.

qnx_crypt()

Encrypt a password (QNX 4)

Synopsis:

```
#include <unistd.h>

char* qnx_crypt( const char* key,
                 const char* salt );
```

Arguments:

key

A NUL-terminated string (normally a password typed by a user).

salt

A two-character string chosen from the set [a-zA-Z0-9./]. This function doesn't validate the values for *salt*, and values outside this range may cause undefined behavior. This string is used to perturb the algorithm in one of 4096 different ways.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `qnx_crypt()` function performs password encryption. It's a variant of the standard [crypt\(\)](#) (p. 385) function that uses an encryption similar to, but not compatible with, the Data Encryption Standard (DES) encryption. This function is provided for compatibility with QNX 4.



The `qnx_crypt()` function checks only the first eight characters of *key*.

Returns:

A pointer to the encrypted value, or NULL on failure.

Examples:

```
#include <unistd.h>

int main(int argc, char **argv) {
    char salt[3];
    char string[20];
    char *result;

    strcpy(string, "thomasf");
    salt[0] = 'a';
    salt[1] = 'B';
    salt[2] = 0;

    result = qnx_crypt(string, salt);
    printf("Result is [%s] --> [%s] \n", string, result);

    return 0;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The return value points to static data that's overwritten by each call to *qnx_crypt()*.

qsort()

Sort an array

Synopsis:

```
#include <stdlib.h>

void qsort( void* base,
            size_t num,
            size_t width,
            int (*compare) (
                const void*,
                const void* ) );
```

Arguments:

base

A pointer to the array that you want to sort.

num

The number of elements in the array.

width

The size of each element, in bytes.

compare

A pointer to a function that compares two entries. It's called with two arguments that point to elements in the array. The comparison function must return an integer less than, equal to, or greater than zero if the first argument is less than, equal to, or greater than the second argument.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `qsort()` function sorts the *base* array using the comparison function specified by *compare*. The array must have at least *num* elements, each of *width* bytes.

Examples:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* some_strs[] = { "last", "middle", "first" };

int compare( const void* op1, const void* op2 )
{
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;

    return( strcmp( *p1, *p2 ) );
}

int main( void )
{
    qsort( some_strs,
          sizeof( some_strs ) / sizeof( char * ),
          sizeof(char *),
          compare );

    printf( "%s %s %s\n",
           some_strs[0], some_strs[1], some_strs[2] );

    return EXIT_SUCCESS;
}

```

produces the output:

```
first last middle
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 20

R

The functions, macros, and data structures in the C library are described here in alphabetical order:

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<code>resmgr_attr_t</code>	resmgr_attach() (p. 2580)
<code>RMSK_CLR()</code>	ThreadCtl() (p. 3408)
<code>RMSK_ISSET()</code>	ThreadCtl() (p. 3408)
<code>RMSK_SET()</code>	ThreadCtl() (p. 3408)
<code>RMSK_SIZE()</code>	ThreadCtl() (p. 3408)

The following have been deprecated:

Instead of:	Use:
<code>rsrdbmgr_minor_attach()</code>	rsrdbmgr_devno_attach() (p. 2688)
<code>rsrdbmgr_minor_detach()</code>	rsrdbmgr_devno_detach() (p. 2691)
<code>rsrdbmgr_query()</code>	rsrdbmgr_query_name() (p. 2693)

This chapter describes the API components whose names begin with “R”.

Raccept()

Accept a connection on a socket (via a SOCKS server)

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int Raccept( int s,
             struct sockaddr * addr,
             int * addrlen );
```

Arguments:

s

A socket that's been created with [socket\(\)](#) (p. 3075).

addr

A result parameter that's filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the connection was made.

addrlen

A value-result parameter. It should initially contain the amount of space pointed to by *addr*; on return it contains the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

Library:

`libsocks`

Use the `-l socks` option to `qcc` to link against this library.

Description:

The *Raccept()* function is a cover function for [accept\(\)](#) (p. 58) — the difference is that *Raccept()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

A descriptor for the accepted socket, or -1 if an error occurs ([errno](#) (p. 539) is set).

Classification:[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

raise()

Generate a signal

Synopsis:

```
#include <signal.h>

int raise( int condition );
```

Arguments:

condition

The signal that you want to raise. For more information, see [signal\(\)](#) (p. 2974).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *raise()* function generates the signal specified by *condition*. Calling *raise()* is equivalent to calling:

```
pthread_kill(pthread_self(), condition);
```

Use [SignalAction\(\)](#) (p. 2977) or [signal\(\)](#) (p. 2974) to specify the actions to take when a signal is received.

Returns:

0 if the specified *condition* is sent, or nonzero if an error occurs ([errno](#) (p. 539) is set).

The *raise()* function doesn't return if the action for that signal is to terminate the program or to transfer control using the [longjmp\(\)](#) (p. 1549) function.

Errors:

EAGAIN

Insufficient system resources are available to deliver the signal.

EINVAL

The value of *condition* isn't a valid signal number.

Examples:

Wait until a SIGINT signal is received. The signal is automatically raised on iteration 10000, or when you press **Ctrl-C**:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

sig_atomic_t signal_count;
sig_atomic_t signal_number;

void alarm_handler( int signum )
{
    ++signal_count;
    signal_number = signum;
}

int main( void )
{
    unsigned long i;

    signal_count = 0;
    signal_number = 0;
    signal( SIGINT, alarm_handler );

    printf("Iteration:      ");
    for( i = 0; i < 100000; ++i ) {
        printf( "\b\b\b\b\b\b%d", 5, i );

        if( i == 10000 ) raise( SIGINT );

        if( signal_count > 0 ) break;
    }

    if( i == 100000 ) {
        printf( "\nNo signal was raised.\n" );
    } else if( i == 10000 ) {
        printf( "\nSignal %d was raised by the "
               "raise() function.\n", signal_number );
    } else {
        printf( "\nUser raised signal #d.\n",
               signal_number );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

rand()

Generate a pseudo-random integer

Synopsis:

```
#include <stdlib.h>

int rand( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *rand()* function computes a pseudo-random integer in the range 0 to RAND_MAX. You can start the sequence at different values by calling [srand\(\)](#) (p. 3155).

The [rand_r\(\)](#) (p. 2486) function is a thread-safe version of *rand()*.

Returns:

A pseudo-random integer.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int i;

    for( i=1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	No

rand_r()

Generate a pseudo-random integer in a thread-safe manner

Synopsis:

```
#include <stdlib.h>

int rand_r( unsigned int* seed );
```

Arguments:

seed

A pointer to the seed for the sequence of pseudo-random numbers. If you call *rand_r()* with the same initial value for the *seed*, the same sequence is generated.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *rand_r()* function computes, in a thread-safe manner, a sequence of pseudo-random integers in the range 0 through RAND_MAX.

Returns:

A pseudo-random integer.

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

random()

Generate a pseudo-random number from the default state

Synopsis:

```
#include <stdlib.h>

long random( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `random()` function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random-number generator is approximately $16 \times (2^{31}-1)$. The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

Use this function in conjunction with the following:

[*initstate\(\)*](#) (p. 1133)

Initialize the state of the pseudo-random number generator.

[*setstate\(\)*](#) (p. 2913)

Specify the state of the pseudo-random number generator.

[*srandom\(\)*](#) (p. 3158)

Set the seed used by the pseudo-random number generator.

The `random()` and `srandom()` functions have (almost) the same calling sequence and initialization properties as [*rand\(\)*](#) (p. 2484) and [*srand\(\)*](#) (p. 3155). The difference is that `rand()` produces a much less random sequence. In fact, the low dozen bits generated

by *rand()* go through a cyclic pattern. All the bits generated by *random()* are usable. For example,

```
random() & 01
```

produces a random binary value.

Unlike *srand()*, *srandom()* doesn't return the old seed because the amount of state information used is much more than a single word. The *initstate()* and *setstate()* routines are provided to deal with restarting/changing random number generators. With 256 bytes of state information, the period of the random-number generator is greater than 2⁶⁹.

Like *rand()*, *random()* produces by default a sequence of numbers that can be duplicated by calling *srandom()* with 1 as the seed.

If *initstate()* hasn't been called, *random()* behaves as though *initstate()* had been called with *seed*=1 and *size*=128.

If *initstate()* is called with *size* less than 8, *random()* uses a simple linear congruential random number generator.

Returns:

The generated pseudo-random number.

Examples:

See [initstate\(\)](#) (p. 1133).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

Rbind()

Bind a name to a socket (via a SOCKS server)

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int Rbind( int s,
           const struct sockaddr * name,
           int namelen );
```

Arguments:

s

The file descriptor to be bound.

name

A pointer to the `sockaddr` structure that holds the address to be bound to the socket. The socket length and format depend upon its address family.

namelen

The length of the `sockaddr` structure pointed to by *name*.

Library:

libsocks

Use the `-l socks` option to `qcc` to link against this library.

Description:

The *Rbind()* function is a cover function for [bind\(\)](#) (p. 208) — the difference is that *Rbind()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Classification:

SOCKS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

rcmd(), rcmd_af()

Execute a command on a remote host

Synopsis:

```
#include <unistd.h>

int rcmd( char ** ahost,
          unsigned short inport,
          const char * locuser,
          const char * remuser,
          const char * cmd,
          int * fd2p );

int rcmd_af( char ** ahost,
             unsigned short inport,
             const char * locuser,
             const char * remuser,
             const char * cmd,
             int * fd2p,
             int af );
```

Arguments:***ahost***

The name of the host that you want to execute the command on. If the function can find the host, it sets **ahost* to the standard name of the host.

inport

The well-known Internet port on the host, where the server resides.

locuser

The user ID on the local machine.

remuser

The user ID on the remote machine.

cmd

The command that you want to execute.

fd2p

See below.

af

(*rcmd_af()* only) The address family; see <sys/socket.h>. If *af* is AF_UNSPEC or PF_UNSPEC, interpretation of **ahost* obeys the underlying address resolution like DNS.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *rcmd()* function is used by the superuser to execute a command, *cmd*, on a remote machine using an authentication scheme based on reserved port numbers. The *rcmd()* function can handle only IPv4 addresses as the first argument; *rcmd_af()* can handle other types of addresses as well. The rshd server (among others) uses the *rcmd()*, [resvport\(\)](#) (p. 2670), and [ruserok\(\)](#) (p. 2696) functions.

The *rcmd()* and *rcmd_af()* functions look up the host **ahost* by means of [gethostbyname\(\)](#) (p. 872), and return -1 if the host doesn't exist. Otherwise, **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a SOCK_STREAM socket in the Internet domain is returned to the caller and given to the remote command as standard input and standard output.

If <i>fd2p</i> is:	Then:
Nonzero	An auxiliary channel to a control process is set up, and a descriptor for it is placed in <i>*fd2p</i> . The control process will return diagnostic output from the command (unit 2) on this channel and will accept bytes as signal numbers to be forwarded to the command's process group.
Zero	The standard error (unit 2 of the remote command) is made the same as the standard output and no provision is made for sending arbitrary signals to the remote process (although you may be able to get its attention by using out-of-band data).

The protocol is described in detail in rshd in the *Utilities Reference*.

Returns:

A valid socket descriptor; or -1 if an error occurs and a message is printed to standard error.

Errors:

The error code EAGAIN is overloaded to mean “All network ports in use.”

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Rconnect()

Initiate a connection on a socket (via a SOCKS server)

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int Rconnect( int s,
              const struct sockaddr * name,
              int namelen);
```

Arguments:

s

The descriptor of the socket on which to initiate the connection.

name

The name of the socket to connect to for a SOCK_STREAM connection.

namelen

The length of the *name*, in bytes.

Library:

libsocks

Use the -l socks option to gcc to link against this library.

Description:

The *Rconnect()* function is a cover function for [connect\(\)](#) (p. 348) — the difference is that *Rconnect()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Classification:[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

rdchk()

Check to see if a read is likely to succeed

Synopsis:

```
#include <unix.h>

int rdchk( int fd );
```

Arguments:

fd

The file descriptor that you want to check.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `rdchk()` function checks to see if a read from the file descriptor, *fd*, is likely to succeed.

Returns:

The number of characters waiting to be read, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

ENOTTY

The *fd* argument isn't the file descriptor for a character device.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

re_comp()

Compile a regular expression

Synopsis:

```
#include <unix.h>

char *re_comp( char *s );
```

Arguments:

s

A string that contains the regular expression that you want to compile. This string must end with a null byte and may include newline characters. If this argument is `NULL`, the current regular expression remains unchanged.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `re_comp()` function converts a regular expression string (RE) into an internal form suitable for pattern matching. Use this function with [re_exec\(\)](#) (p. 2500).

The `re_comp()` and `re_exec()` functions support simple regular expressions. The regular expressions of the form `\{m\}`, `\{m, \}`, or `\{m, n\}` aren't supported.



For better portability, use [regcomp\(\)](#) (p. 2542), [regerror\(\)](#) (p. 2547), and [regexexec\(\)](#) (p. 2549) instead of these functions.

Returns:

`NULL` if the string pointed to by `s` was successfully converted. Otherwise, a pointer to one of the following error message strings is returned:

- No previous regular expression

- Regular expression too long
- unmatched \((
- missing]
- too many \(\) pairs
- unmatched \)

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

re_exec()

Execute a regular expression

Synopsis:

```
#include <unix.h>

int re_exec( char *s );
```

Arguments:

s

A pointer to the string that you want to compare to the current regular expression. This string must end with a null byte and may include newline characters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `re_exec()` function compares the string pointed to by the string argument with the last regular expression passed to [re_comp\(\)](#) (p. 2498).

The `re_comp()` and `re_exec()` functions support simple regular expressions. The regular expressions of the form `\{m\}`, `\{m, \}`, or `\{m, n\}` aren't supported.



For better portability, use [regcomp\(\)](#) (p. 2542), [regerror\(\)](#) (p. 2547), and [regexexec\(\)](#) (p. 2549) instead of these functions.

Returns:

1

The string matches the last compiled regular expression.

0

The string doesn't match the last compiled regular expression.

-1

The compiled regular expression is invalid (indicating an internal error).

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

read()

Read bytes from a file

Synopsis:

```
#include <unistd.h>

ssize_t read( int fildes,
              void* buf,
              size_t nbyte );
```

Arguments:

fildes

The descriptor of the file that you want to read from.

buf

A pointer to a buffer where the function can store the data that it reads.

nbyte

The number of bytes that you want to read.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *read()* function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*.

If *nbyte* is zero, *read()* returns zero, and has no other effect.

On a regular file or other file capable of seeking, *read()* starts at a position in the file given by the file offset associated with *fildes*. Before successfully returning from *read()*, the file offset is incremented by the number of bytes actually read.



The *read()* call ignores advisory locks that you may have set with *fcntl()* (p. 623).

On a file not capable of seeking, *read()* starts at the current position.

When *read()* returns successfully, its return value is the number of bytes actually read and placed in the buffer. This number will never be greater than *nbyte*, although it may be less than *nbyte* for one of the following reasons:

- The number of bytes left in the file is less than *nbyte*.
- The *read()* request was interrupted by a signal.
- The file is a pipe (or FIFO) or a special file, and has fewer than *nbyte* bytes immediately available for reading. For example, reading from a file associated with a terminal may return one typed line of data.

If *read()* is interrupted by a signal before it reads any data, it returns a value of -1 and sets *errno* (p. 539) to EINTR. However, if *read()* is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

No data is transferred past the current end-of-file. If the starting position is at or after the end-of-file, *read()* returns zero. If the file is a device special file, the result of subsequent calls to *read()* will work, based on the then current state of the device (that is, the end of file is transitory).

If the value of *nbyte* is greater than INT_MAX, *read()* returns -1 and sets *errno* to EINVAL. See <limits.h>.

When attempting to read from an empty pipe or FIFO:

1. If no process has the pipe open for writing, *read()* returns 0 to indicate end-of-file.
2. If a process has the pipe open for writing, and O_NONBLOCK is set, *read()* returns -1, and *errno* is set to EAGAIN.
3. If a process has the pipe open for writing, and O_NONBLOCK is clear, *read()* blocks until some data is written, or the pipe is closed by all processes that had opened it for writing.

When attempting to read from a file (other than a pipe or FIFO) that support nonblocking reads and has no data currently available:

1. If O_NONBLOCK is set, *read()* returns -1, and *errno* is set to EAGAIN.
2. If O_NONBLOCK is clear, *read()* blocks until some data is available.
3. The O_NONBLOCK flag has no effect if some data is available.

If you call *read()* on a portion of a file, prior to the end-of-file, that hasn't been written, it returns bytes with the value zero.

If *read()* succeeds, the *st_atime* field of the file is marked for update.

Returns:

The number of bytes actually read, or -1 (*errno* (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the read operation.

EBADF

The file descriptor, *fdes*, isn't a valid file descriptor open for reading.

EINTR

The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EIO

One of the following:

- A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.
- The filesystem resides on a removable media device, and the media has been forcibly removed.

ENOSYS

The *read()* function isn't implemented for the filesystem specified by *files*.

EOVERFLOW

The file is a regular file and an attempt is made to read at or beyond the offset maximum associated with the file.

Examples:

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int fd;
    int size_read;
    char buffer[80];

    /* Open a file for input */
    fd = open( "myfile.dat", O_RDONLY );

    /* Read the text */
    size_read = read( fd, buffer,
                     sizeof( buffer ) );

    /* Test for error */
    if( size_read == -1 ) {
        perror( "Error reading myfile.dat" );
        return EXIT_FAILURE;
    }
}
```

```
    /* Close the file */
    close( fd );

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

readblock()

Read blocks of data from a file

Synopsis:

```
#include <unistd.h>

int readblock( int fd,
               size_t blksize,
               unsigned block,
               int numblks,
               void *buff );
```

Arguments:

fd

The file descriptor for the file you want to read from.

blksize

The number of bytes in each block of data.

block

The block number from which to start reading.

numblks

The number of blocks to read.

buff

A pointer to a buffer where the function can store the data that it reads.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *readblock()* function reads *numblks* blocks of data from the file associated with the open file descriptor *fd*, into the buffer pointed to by *buff*, starting at block number *block* (blocks are numbered starting at 0). The *blksize* argument specifies the size of a block, in bytes.

This function is useful for direct access to raw blocks on a block special device (for example, raw disk blocks) but may also be used for high-speed access to database files, for example. (The speed gain is through the combined seek/read implicit in this call, and the ability to transfer more than the *read()* function's limit of `INT_MAX` bytes at a time.)

If *numblks* is zero, *readblock()* returns zero and has no other results.

On successful completion, *readblock()* returns the number of blocks actually read and placed in the buffer. This number is never greater than *numblks*. The value returned may be less than *numblks* if one of the following occurs:

- The number of blocks left before the end-of-file is less than *numblks*.
- The process requests more blocks than implementation limits allow to be read in a single atomic operation.
- A read error occurred after reading at least one block.

If a read error occurs on the first block, *readblock()* returns -1 and sets *errno* to `EIO`.

Returns:

The number of blocks actually read. If an error occurs, it returns -1, sets *errno* (p. 539) to indicate the error, and the contents of the buffer pointer to by *buff* are left unchanged.

Errors:

EBADF

The *fd* argument isn't a valid file descriptor open for reading a block-oriented device.

EIO

A physical read error occurred on the first block.

EINVAL

The starting position is invalid (0 or negative) or beyond the end of the disk.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

readcond()

Read data from a terminal device

Synopsis:

```
#include <unistd.h>

int readcond( int fd,
              void * buf,
              int n,
              int min,
              int time,
              int timeout );
```

Arguments:

fd

The file descriptor associated with the terminal device that you want to read from.

buf

A pointer to a buffer into which *readcond()* can put the data.

n

The maximum number of bytes to read.

min, time, timeout

When used in RAW mode, these arguments override the behavior of the *MIN* and *TIME* members of the terminal's [termios](#) (p. 3379) structure. For more information, see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *readcond()* function reads up to *n* bytes of data from the terminal device indicated by *fd* into the buffer pointed to by *buf*.

This function is an alternative to the [read\(\)](#) (p. 2502) function for terminal devices, providing additional arguments for timed read operations. These additional arguments can be used to minimize overhead when dealing with terminal devices.

The three arguments (*min*, *time*, and *timeout*), when used on terminal devices in RAW mode, override the behavior of the *MIN* and *TIME* elements of the currently defined [termios](#) (p. 3379) structure (for the duration of this call only). The *termios* structure also defines a forwarding character (in *c_cc[VFWD]*) that can be used to bypass *min*, *time* and *timeout*.

The normal case of a simple read by an application would block until at least one character was available.

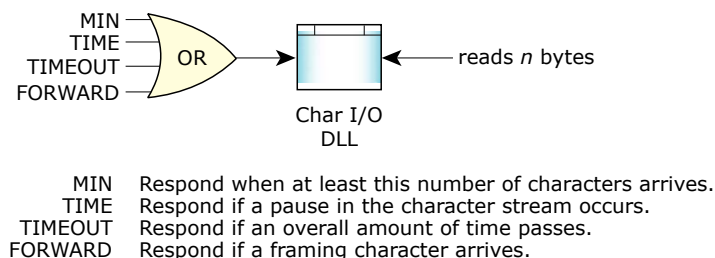


Figure 9: Conditions that satisfy an input request.

In the case where multiple conditions are specified, the read will be satisfied when any one of them is satisfied.

MIN

The qualifier *MIN* is useful when an application has knowledge of the number of characters it expects to receive.

Any protocol that knows the character count for a frame of data can use *MIN* to wait for the entire frame to arrive. This significantly reduces IPC and process scheduling. *MIN* is often used in conjunction with *TIME* or *TIMEOUT*. *MIN* is part of the POSIX standard.

TIME

The qualifier *TIME* is useful when an application is receiving streaming data and wishes to be notified when the data stops or pauses. The pause time is specified in tenths of a second. *TIME* is part of the POSIX standard.

TIMEOUT

The qualifier *TIMEOUT* is useful when an application has knowledge of how long it should wait for data before timing out. The timeout is specified in tenths of a second.

Any protocol that knows the character count for a frame of data it expects to receive can use *TIMEOUT*. This in combination with the baud rate allows a reasonable guess to be made when data should be available. It acts as a deadman timer to detect

dropped characters. It can also be used in interactive programs with user input to time out a read if no response is available within a given time.

TIMEOUT is a QNX Neutrino extension and isn't part of the POSIX standard.

FORWARD

The qualifier *FORWARD* is useful when a protocol is delimited by a special framing character. For example, the PPP protocol used for TCP/IP over a serial link start and end its packets with a framing character. When used in conjunction with *TIMEOUT*, the *FORWARD* character can greatly improve the efficiency of a protocol implementation. The protocol process will receive complete frames, rather than character by character. In the case of a dropped framing character, *TIMEOUT* or *TIME* can be used to quickly recover.

This greatly minimizes the amount of IPC work for the OS and results in a much lower processor utilization for a given TCP/IP data rate. It is interesting to note that PPP doesn't contain a character count for its frames. Without the data-forwarding character, an implementation would be forced to read the data one character at a time.

FORWARD is a QNX Neutrino extension and isn't part of the POSIX standard.

To enable the *FORWARD* character, you must set the VFWD character in the *c_cc* member of the *termios* structure:

```
/* PPP forwarding character */
const char fwd_char = 0x7e;

#include <termios.h>

...

int fd;
struct termios termio;

...

tcgetattr( fd, &termio );
termio.c_cc[VFWD] = fwd_char;
tcsetattr( fd, TCSANOW, &termio );
```

The following table summarizes the interaction of *min*, *time*, and *timeout*:

<i>min</i>	<i>time</i>	<i>timeout</i>	Description
0	0	0	Returns immediately with as many bytes as are currently available (up to <i>n</i> bytes).
<i>M</i>	0	0	Return with up to <i>n</i> bytes only when at least <i>M</i> bytes are available.

<i>min</i>	<i>time</i>	<i>timeout</i>	Description
0	T	0	Return with up to n bytes when at least one byte is available, or $T * .1$ sec has expired.
M	T	0	Return with up to n bytes when either M bytes are available, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds $T * .1$ sec.
0	0	t	RESERVED.
M	0	t	Return with up to n bytes when $t * .1$ sec has expired, or M bytes are available.
0	T	t	RESERVED.
M	T	t	Return with up to n bytes when M bytes are available, or $t * .1$ sec has expired and no characters are received, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds $T * .1$ sec.

Note that when *timeout* is zero, the behavior of *min* and *time* is exactly the same as the behavior of the MIN and TIME parameters of the `termios` controlling structure. Thus, [*readcond\(\)*](#) (p. 2509) can be used as a higher speed alternative to consecutive

calls of [tcgetattr\(\)](#) (p. 3347), [tcsetattr\(\)](#) (p. 3364), and [read\(\)](#) (p. 2502) when dealing with RAW terminal I/O.

The $(M, 0, t)$ case is useful for communications protocols that cannot afford to block forever waiting for data that may never arrive.

The (M, T, t) case is provided to permit [readcond\(\)](#) to return when a burst of data ends (as in the $(M, T, 0)$ case), but also to return if no burst at all is detected within a reasonable amount of time.

Returns:

The number of bytes read, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set on this *fd*, and the process would have been blocked in trying to perform this operation.

EBADF

The argument *fd* is invalid or file isn't opened for reading.

EINTR

The [readcond\(\)](#) call was interrupted by the process being signalled.

EIO

This process isn't currently able to read data from this *fd*.

ENOSYS

This function isn't supported for this *fd*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

readdir(), *readdir64()*

Read a directory entry

Synopsis:

```
#include <dirent.h>

struct dirent * readdir( DIR * dirp );
struct dirent64 * readdir64( DIR *dirp );
```

Arguments:

dirp

A pointer to the directory stream to be read.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *readdir()* and *readdir64()* functions read the next directory entry from the directory specified by *dirp*, which is the value returned by a call to [opendir\(\)](#) (p. 1913). The *readdir64()* function is a large-file version of *readdir()*.



The [ftw\(\)](#) (p. 798) function provides another way to walk a file tree.

You can call *readdir()* repeatedly to list all of the entries contained in the directory specified by the pathname given to *opendir()*. You must call [closedir\(\)](#) (p. 338) to close the directory stream and free the memory allocated by *opendir()*.

The `<dirent.h>` file defines the [dirent](#) and [dirent64](#) (p. 422) structures, and the `DIR` type used by the *readdir()* family of functions.



The result of using a directory stream after one of the [exec*\(\)](#) (p. 550) or [spawn*\(\)](#) (p. 3089) family of functions is undefined. After a call to [fork\(\)](#) (p. 685), either the parent or the child (but not both) can continue processing the directory stream, using the *readdir()* and *rewinddir()* functions. If both the

parent and child processes use these functions, the result is undefined. Either (or both) processes may use [closedir\(\)](#) (p. 338).

The `<dirent.h>` file also defines the following macros for accessing extra data associated with the `dirent` and `dirent64` structures:

`_DEXTRA_FIRST( pdirent )`

Get a pointer to the first block of data associated with the structure pointed to by `pdirent`.

`_DEXTRA_NEXT( last)`

Get the block of data that follows the block pointed to by `last`.

`_DEXTRA_VALID( extra, pdirent)`

Evaluates to 1 if `extra` is a pointer to a valid block of data associated with the structure pointed to by `pdirent`.

The blocks of data are `dirent_extra` or `dirent_extra_stat` structures:

```
struct dirent_extra {
    uint16_t      d_dataalen;      /* data size (without header) */
    uint16_t      d_type;
    uint32_t      d_reserved;
    /* unsigned char d_data[d_dataalen] */
};

struct dirent_extra_stat {
    uint16_t      d_dataalen;      /* sizeof(struct stat) */
    uint16_t      d_type;
    uint32_t      d_reserved;
    struct stat    d_stat;
};
```

You can use these macros to traverse the data associated with the `dirent` structure like this:

```
for( extra = _DEXTRA_FIRST(dirent);
    _DEXTRA_VALID(extra, dirent);
    extra = _DEXTRA_NEXT(extra)) {
    switch(extra->d_type) {
        /* No data */
        case _DTYPE_NONE :
            break;
        /* Data includes information as returned by stat() */
        case _DTYPE_STAT :
            break;
        /* Data includes information as returned by lstat() */
        case _DTYPE_LSTAT :
            break;
        ...
    }
}
```

If the value of the `d_type` member is `_DTYPE_STAT` or `_DTYPE_LSTAT`, the `d_stat` member points to a [stat structure](#) (p. 3164) (see the entry for [stat\(\)](#) (p. 3164) in this reference). You can use the [dircntl\(\)](#) (p. 419) function to request that the filesystem include this extra information.

Returns:

A pointer to a `struct dirent` or `struct dirent64` object for success, or `NULL` if the end of the directory stream is encountered or an error occurs ([errno](#) (p. 539) is set).



- A returned value of `NULL` is ambiguous; if you need to determine if it indicates an error (as opposed to the end of the directory stream), set `errno` to `EOK` before each call to this function.
- Although `dirent` and `dirent64` are declared with a one-byte `d_name` field, the structure that `readdir()` and `readdir64()` returns is allocated with enough space to hold the entire name.
- Subsequent calls to `readdir()` or `readdir64()` with the same directory stream may overwrite the memory that the returned pointer points to.

Errors:**EBADF**

The `dirp` argument doesn't refer to an open directory stream.

EOVERFLOW

One of the values in the structure to be returned can't be represented correctly.

Examples:

Get a list of files contained in the directory `/home/fred`:

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

int main( void )
{
    DIR* dirp;
    struct dirent* direntp;

    dirp = opendir( "/home/fred" );
    if( dirp != NULL ) {
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;

            printf( "%s\n", direntp->d_name );
        }

        closedir( dirp );
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```


Classification:

readdir() is [POSIX 1003.1](#) (p. 46); *readdir64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

readdir_r()

Get information about the next matching filename

Synopsis:

```
#include <sys/types.h>
#include <dirent.h>

int readdir_r( DIR * dirp,
               struct dirent * entry,
               struct dirent ** result );
```

Arguments:

dirp

A pointer to the directory stream to be read.

entry

A pointer to a [dirent](#) (p. 422) structure where the function can store the directory entry.

result

The address of a location where the function can store a pointer to the information found.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, `readdir_r()` initializes the `dirent` structure referenced by `entry` with the directory entry at the current position in the directory stream referred to by `dirp`, and stores a pointer to this structure in `result`.

The storage pointed by `entry` must be large enough for a `dirent` structure with the `s_name` member an array of `char` containing at least `NAME_MAX` plus one element.

The `struct dirent` structure *doesn't* include space for the pathname; you must provide it. For example:



```
struct dirent *entry;
entry = malloc( offsetof(struct dirent, d_name) + NAME_MAX + 1 );
```

Some filesystems support names that are longer than the value of `NAME_MAX`. You can use [*pathconf\(\)*](#) (p. 1936) with `_PC_NAME_MAX` to determine the maximum number of bytes (not including the terminating null) allowed in a file name for a particular filesystem.

Returns:

EOK

Success.

EBADF

The *dirp* argument doesn't refer to an open directory stream.

E_OVERFLOW

One of the values in the structure to be returned can't be represented correctly.

Classification:

[*POSIX 1003.1 TSF*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

readlink()

Place the contents of a symbolic link into a buffer

Synopsis:

```
#include <unistd.h>

int readlink( const char* path,
              char* buf,
              size_t bufsiz );
```

Arguments:

path

The name of the symbolic link.

buf

A pointer to a buffer where the function can store the contents of the symbolic link (i.e. the path linked to).

bufsiz

The size of the buffer.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *readlink()* function places the contents of the symbolic link named by *path* into the buffer pointed to by *buf*, which has a size of *bufsiz*. The contents of the returned symbolic link doesn't include a `NULL` terminator. Its length must be determined from the `stat` structure returned by *lstat()* (p. 1558), or by the return value of the *readlink()* call.

If *readlink()* is successful, up to *bufsiz* bytes from the contents of the symbolic link are placed in *buf*.

Returns:

The number of bytes placed in the buffer, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EACCES**

Search permission is denied for a component of the *path* prefix.

EINVAL

The named file isn't a symbolic link.

ELOOP

A loop exists in the symbolic links encountered during resolution of the path argument, and more than SYMLoop_MAX symbolic links were encountered.

ENAMETOOLONG

A component of the *path* exceeded NAME_MAX characters, or the entire pathname exceeded PATH_MAX characters.

ENOENT

The named file doesn't exist.

ENOSYS

Links aren't supported by the resource manager associated with path.

ENOTDIR

A component of the *path* prefix named by path isn't a directory.

Examples:

```
/*
 * Read the contents of the named symbolic links
 */
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char buf[PATH_MAX + 1];

int main( int argc, char** argv )
{
    int n;
    int len;
    int ecode = 0;

    for( n = 1; n < argc; ++n ) {
        if( ( len = readlink( argv[n], buf, PATH_MAX ) ) == -1 ) {
            perror( argv[n] );
            ecode++;
        }
        else {
            buf[len] = '\0';
            printf( "%s -> %s\n", argv[n], buf );
        }
    }

    return( ecode );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

readv()

Read bytes from a file

Synopsis:

```
#include <sys/uio.h>

ssize_t readv( int fildes,
               const iove_t* iov,
               int iovcnt );
```

Arguments:

fildes

The descriptor of the file that you want to read from.

iov

An array of *iovec_t* objects where the function can store the data that it reads.

iovcnt

The number of entries in the *iov* array. The maximum number of entries is `UIO_MAXIOV`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *readv()* function attempts to read from the file associated with the open file descriptor, *fildes*, placing the data into *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1].

On a regular file or other file capable of seeking, *readv()* starts at a position in the file given by the file offset associated with *fildes*. Before successfully returning from *readv()*, the file offset is incremented by the number of bytes actually read.

The *iovec_t* structure contains the following members:

iov_base

The base address of a memory area where data should be placed.

iov_len

The length of the memory area.

The *readv()* function always fills one buffer completely before proceeding to the next.



The *readv()* call ignores advisory locks that may have been set by the *fcntl()* (p. 623) function.

On a file not capable of seeking, *readv()* starts at the current position.

When *readv()* returns successfully, its return value is the number of bytes actually read and placed in the buffer. This number will never be greater than the combined sizes of the *iov* buffers, although it may be less for one of the following reasons:

- The number of bytes left in the file is less than the combined size of the *iov* buffers.
- The *readv()* request was interrupted by a signal.
- The file is a pipe (or FIFO) or a special file, and has fewer bytes immediately available for reading. For example, reading from a file associated with a terminal may return one typed line of data.

If *readv()* is interrupted by a signal before it reads any data, it returns a value of -1 and sets *errno* (p. 539) to *EINTR*. However, if *readv()* is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

No data is transferred past the current end-of-file. If the starting position is at or after the end-of-file, *readv()* returns zero. If the file is a device special file, the result of subsequent calls to *readv()* will work, based on the then current state of the device (that is, the end of file is transitory).

When attempting to read from an empty pipe or FIFO:

1. If no process has the pipe open for writing, *readv()* returns 0 to indicate end-of-file.
2. If a process has the pipe open for writing, and *O_NONBLOCK* is set, *readv()* returns -1 and sets *errno* to *EAGAIN*.
3. If a process has the pipe open for writing, and *O_NONBLOCK* is clear, *readv()* blocks until some data is written, or the pipe is closed by all processes that had opened it for writing.

When attempting to read from a file (other than a pipe or FIFO) that supports nonblocking reads and has no data currently available:

1. If *O_NONBLOCK* is set, *readv()* returns -1 and sets *errno* to *EAGAIN*.
2. If *O_NONBLOCK* is clear, *readv()* blocks until some data is available.
3. The *O_NONBLOCK* flag has no effect if some data is available.

If you call *readv()* on a portion of a file, prior to the end-of-file, that hasn't been written, it returns bytes with the value zero.

If *readv()* succeeds, the *st_atime* field of the file is marked for update.

Returns:

The number of bytes read, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EAGAIN

The *O_NONBLOCK* flag is set for the file descriptor, and the process would be delayed in the read operation.

EBADF

The file descriptor, *fildev*, isn't a valid file descriptor open for reading.

EINTR

The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EINVAL

The *iovcnt* argument is less than or equal to 0, or greater than *UIO_MAXIOV*.

EIO

A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSYS

The *readv()* function isn't implemented for the filesystem specified by *filedev*.

EOVERFLOW

The file is a regular file and an attempt is made to read at or beyond the offset maximum associated with the file.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

realloc()

Allocate, reallocate, or free a block of memory

Synopsis:

```
#include <stdlib.h>

void* realloc( void* old_blk,
               size_t size );
```

Arguments:

old_blk

A pointer to the block of memory to be allocated, reallocated, or freed.

size

The new size, in bytes, for the block of memory.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *realloc()* function allocates, reallocates, or frees the block of memory specified by *old_blk* based on the following rules. You can control parts of this behavior via the ***MALLOC_OPTIONS*** environmental variable.

- If *old_blk* is `NULL` and *size* isn't 0, *realloc()* allocates a new block of memory of *size* bytes.
- If *old_blk* is `NULL` and *size* is 0, the default behavior is to return a non-`NULL` pointer that's valid only to a corresponding call to [free\(\)](#) (p. 717) or *realloc()*. Don't assume that this pointer points to any valid memory. If the value of ***MALLOC_OPTIONS*** contains a `V`, *realloc()* returns a `NULL` pointer. This environment variable also affects [calloc\(\)](#) (p. 255) and [malloc\(\)](#) (p. 1569). This is known as the “System V” behavior.
- If *old_blk* is non-`NULL` and *size* is 0, the default behavior is to return a valid non-`NULL` pointer that's valid only to a corresponding call to *free()* or *realloc()*. Don't assume that this pointer points to any valid memory.



The default behavior in this case changed in QNX Neutrino 6.4.1.

If the value of **MALLOC_OPTIONS** contains an R or a V, *realloc()* calls *free()* to release the memory pointed to by *old_blk*, and then returns NULL. This is the way QNX Neutrino 6.4.0 and earlier behaved.

- Otherwise, *realloc()* reallocates space for an object of *size* bytes by:
 - shrinking the size of the allocated memory block *old_blk* when *size* is smaller than the current size of *old_blk*
 - extending the allocated size of the allocated memory block *old_blk* if there is a large enough block of unallocated memory immediately following *old_blk*
 - allocating a new block with the appropriate *size*, and copying the contents of *old_blk* to this new block

The *realloc()* function allocates memory from the heap. Use [free\(\)](#) (p. 717) or *realloc()* to free the block of memory.



- The *malloc()* implementation uses signed, 32-bit integers to represent the size internally, so you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, *realloc()* indicates an error of ENOMEM.
- Because it's possible that a new block will be allocated, any pointers into the old memory could be invalidated. These pointers will point to freed memory, with possible disastrous results, when a new block is allocated.

The *realloc()* function returns NULL when the memory pointed to by *old_blk* can't be reallocated. In this case, the memory pointed to by *old_blk* isn't freed, so be careful to maintain a pointer to the old memory block so you can free it later.

In the following example, *buffer* is set to NULL if the function fails, and won't point to the old memory block. If *buffer* is your only pointer to the memory block, then you have lost access to this memory.

```
buffer = (char* )realloc( buffer, 100 );
```

Returns:

A pointer to the start of the allocated memory, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

ENOMEM

Not enough memory.

EOK

No error.

Examples:

```
#include <stdlib.h>
#include <malloc.h>

int main( void )
{
    char* buffer;
    char* new_buffer;

    buffer = (char* )malloc( 80 );
    if ( buffer == NULL ) {
        return EXIT_FAILURE;
    }

    new_buffer = (char* )realloc( buffer, 100 );
    if( new_buffer == NULL ) {

        /* Couldn't allocate a larger buffer.
         * Remember that buffer stills point to
         * allocated memory -- don't leak it! */

        free (buffer);
        return EXIT_FAILURE;
    } else {
        buffer = new_buffer;
    }

    return EXIT_SUCCESS;
}
```

Environment variables:**MALLOC_OPTIONS**

Control the way *calloc()*, *malloc()*, and *realloc()* behave if you specify a size of 0 (or a value of 0 for the *n* argument to *calloc()*). The V (“System V”) and R (“use the *realloc()* behavior of QNX Neutrino 6.4.0 and earlier”) columns below indicate how the functions behave if the value of **MALLOC_OPTIONS** includes that letter:

Function	Default	V	R
<i>calloc</i> (<i>n</i> , 0)	Non-NULL	NULL	No effect
<i>malloc</i> (0)	Non-NULL	NULL	No effect
<i>realloc</i> (NULL, 0)	Non-NULL	NULL	No effect
<i>realloc</i> (non-NULL, 0)	Non-NULL	NULL	NULL

In all the above cases, if the function returns a non-NULL pointer, it's valid only for a corresponding call to *free()* or *realloc()*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

realpath()

Resolve a pathname

Synopsis:

```
#include <stdlib.h>

char * realpath( const char * pathname,
                 char * resolved_name );
```

Arguments:

pathname

The path name that you want to resolve.

resolved_name

NULL, or a pointer to a buffer where the function can store the resolved name.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *realpath()* function resolves all symbolic links, extra slash (/) characters and references to `/./` and `/../` in *pathname*, and copies the resulting null-terminated absolute pathname into the memory referenced by *resolved_name*.

To determine the size of the buffer pointed to by *resolved_name*, call [fpathconf\(\)](#) (p. 699) or [pathconf\(\)](#) (p. 1936) with an argument of `_PC_PATH_MAX`.

If *resolved_name* is NULL, the function uses [malloc\(\)](#) (p. 1569) to allocate a buffer of the size indicated by `_PC_PATH_MAX`. You're responsible for freeing this memory by calling [free\(\)](#) (p. 717).

This function resolves both absolute and relative paths and returns the absolute pathname corresponding to *pathname*. All but the last component of *pathname* must exist when you call *realpath()*.

Returns:

A pointer to *resolved_name*, or NULL if an error occurred, in which case *resolved_name* contains the pathname that caused the problem.

Errors:

The *realpath()* function may fail and set the external variable *errno* (p. 539) to any of the errors specified for the library functions *chdir()* (p. 286), *close()* (p. 336), *fchdir()* (p. 610), *lstat()* (p. 1558), *open()* (p. 1905), *readlink()* (p. 2520) and *getcwd()* (p. 830).

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

recv()

Receive a message from a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv( int s,
              void * buf,
              size_t len,
              int flags );
```

Arguments:

s

The descriptor for the socket; see [socket\(\)](#) (p. 3075).

buf

A pointer to a buffer where the function can store the message.

len

The size of the buffer.

flags

A combination formed by ORing one or more of the values:

- **MSG_OOB** — process out-of-band data. This flag requests receipt of out-of-band data that wouldn't be received in the normal data stream. You can't use this flag with protocols that place expedited data at the head of the normal data queue.
- **MSG_PEEK** — peek at the incoming message. This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
- **MSG_WAITALL** — wait for full request or error. This flag requests that the operation block until the full request is satisfied. But the call may still return less data than requested if a signal is caught, if an error or disconnect occurs, or if the next data to be received is of a different type than that returned.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `recv()` function receives a message from a socket. It's normally used only on a *connected* socket — see [connect\(\)](#) (p. 348) — and is identical to [recvfrom\(\)](#) (p. 2536) with a zero *from* parameter.

This routine returns the length of the message on successful completion. If a message is too long for the supplied buffer, *buf*, then excess bytes might be discarded, depending on the type of socket that the message is received from; see [socket\(\)](#) (p. 3075).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking—see [ioctl\(\)](#) (p. 1198)—in which case `-1` is returned and the external variable *errno* is set to `EWOULDBLOCK`. Normally, the receive calls return any data available, up to the requested amount, rather than wait for the full amount requested; this behavior is affected by the socket-level options `SO_RCVLOWAT` and `SO_RCVTIMEO` described in [getsockopt\(\)](#) (p. 962).

You can use `select()` to determine when more data is to arrive.

Returns:

The number of bytes received, or `-1` if an error occurs (*errno* (p. 539) is set).

Errors:**EBADF**

Invalid descriptor *s*.

EFAULT

The receive buffer is outside the process's address space.

EINTR

The receive was interrupted by delivery of a signal before any data was available.

ENOTCONN

The socket is associated with a connection-oriented protocol and hasn't been connected; see [connect\(\)](#) and [accept\(\)](#) (p. 58).

EWOULDBLOCK

Either the socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

recvfrom()

Receive a message from the socket at a specified address

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom( int s,
                  void * buff,
                  size_t len,
                  int flags,
                  struct sockaddr * from,
                  socklen_t * fromlen );
```

Arguments:

s

The descriptor for the socket; see [socket\(\)](#) (p. 3075).

buf

A pointer to a buffer where the function can store the message.

len

The size of the buffer.

flags

A combination formed by ORing one or more of the values:

- **MSG_OOB** — process out-of-band data. This flag requests receipt of out-of-band data that wouldn't be received in the normal data stream. You can't use this flag with protocols that place expedited data at the head of the normal data queue.
- **MSG_PEEK** — peek at the incoming message. This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
- **MSG_WAITALL** — wait for full request or error. This flag requests that the operation block until the full request is satisfied. But the call may still return less data than requested if a signal is caught, if an error or disconnect occurs, or if the next data to be received is of a different type than that returned.

from

NULL, or a pointer to a `sockaddr` object where the function can store the source address of the message.

fromlen

A pointer to a `socklen_t` object that specifies the size of the *from* buffer. The function stores the actual size of the address in this object.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *recvfrom()* routine receives a message from the socket, *s*, whether or not it's connection-oriented.

If *from* is nonzero, and the socket is connectionless, the source address of the message is filled in. The parameter *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the stored address.

This routine returns the length of the message on successful completion. If a message is too long for the supplied buffer, *buf*, excess bytes may be discarded depending on the type of socket that the message is received from—see [socket\(\)](#) (p. 3075).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking—see [ioctl\(\)](#) (p. 1198)—in which case *recvfrom()* returns -1 is returned and sets the external variable *errno* to `EWOULDBLOCK`. Normally, the receive calls return any data available, up to the requested amount, rather than wait for the full amount requested; this behavior is affected by the socket-level options `SO_RCVLOWAT` and `SO_RCVTIMEO` described in [getsockopt\(\)](#) (p. 962).

You can use [select\(\)](#) (p. 2791) to determine when more data is to arrive.

Returns:

The number of bytes received, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EBADF**

Invalid descriptor *s*.

EFAULT

The receive buffer pointer(s) point outside the process's address space.

EINTR

The receive was interrupted by delivery of a signal before any data was available.

ENOTCONN

The socket is associated with a connection-oriented protocol and hasn't been connected; see *connect()* and *accept()*.

EWOULDBLOCK

Either the socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

recvmsg()

Receive a message and its header from a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvmsg( int s,
                 struct msghdr * msg,
                 int flags );
```

Arguments:

s

The descriptor for the socket; see [socket\(\)](#) (p. 3075).

msg

A pointer to a `msghdr` structure where the function can store the message header; see below.

flags

A combination formed by ORing one or more of the values:

- **MSG_OOB** — process out-of-band data. This flag requests receipt of out-of-band data that wouldn't be received in the normal data stream. You can't use this flag with protocols that place expedited data at the head of the normal data queue.
- **MSG_PEEK** — peek at the incoming message. This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
- **MSG_WAITALL** — wait for full request or error. This flag requests that the operation block until the full request is satisfied. But the call may still return less data than requested if a signal is caught, if an error or disconnect occurs, or if the next data to be received is of a different type than that returned.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `recvmsg()` routine receives a message from a socket, `s`, whether or not it's connection-oriented.

The `recvmsg()` call uses a `msghdr` structure to minimize the number of directly supplied parameters. This structure, defined in `<sys/socket.h>`, has the following form:

```
struct msghdr {
    caddr_t msg_name;           /* optional address */
    u_int   msg_namelen;        /* size of address */
    struct  iovec *msg_iov;      /* scatter/gather array */
    u_int   msg_iovlen;         /* # elements in msg_iov */
    caddr_t msg_control;         /* ancillary data, see below */
    u_int   msg_controllen;     /* ancillary data buffer len */
    int     msg_flags;          /* flags on received message */
};
```

The `msg_name` and `msg_namelen` parameters specify the address (source address for `recvmsg()`; destination address for [sendmsg\(\)](#) (p. 2833)) if the socket is unconnected; the `msg_name` parameter may be given as a null pointer if no names are desired or required.

The `msg_iov` and `msg_iovlen` parameters describe scatter-gather locations, as discussed in [read\(\)](#) (p. 2502).

The `msg_control` parameter, whose length is determined by `msg_controllen`, points to a buffer for other protocol-control related messages or for other miscellaneous ancillary data. The messages are of the form:

```
struct cmsghdr {
    u_int cmsg_len;             /* data byte count, including hdr */
    int   cmsg_level;           /* originating protocol */
    int   cmsg_type;            /* protocol-specific type */
    /* followed by u_char cmsg_data[]; */
};
```

The `msg_flags` field is set on return according to the message received:

MSG_CTRUNC

Indicates that some control data was discarded due to lack of space in the buffer for ancillary data.

MSG_EOR

Indicates end-of-record; the data returned completed a record.

MSG_OOB

Indicates that expedited or out-of-band data was received.

MSG_TRUNC

Indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied.

Returns:

The number of bytes received, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**ENOMEM**

Not enough memory.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

regcomp()

Compile a regular expression

Synopsis:

```
#include <regex.h>

int regcomp( regex_t * preg,
             const char * pattern,
             int cflags );
```

Arguments:

preg

A pointer to a `regex_t` object where the function can store the compiled regular expression.

pattern

The regular expression that you want to compile; see below.

cflags

A bitwise inclusive OR of zero or more of the following flags:

- `REG_EXTENDED` — use Extended Regular Expressions.
- `REG_ICASE` — ignore differences in case.
- `REG_NEWLINE` — treat <newline> as a regular character.
- `REG_NOSUB` — report only success/failure in [regexexec\(\)](#) (p. 2549).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *regcomp()* function prepares the regular expression, *preg*, for use by the function *regexexec()*, from the specification *pattern* and *cflags*. The member *re_nsub* of *preg* is set to the number of subexpressions in *pattern*.

The functions that deal with regular expressions (*regcomp()*, *regerror()* (p. 2547), *regexexec()* (p. 2549), and *regfree()* (p. 2552)) support two classes of regular expressions, the *Basic* (p. 2543) and *Extended Regular* (p. 2544) Expressions. These classes are rigorously defined in IEEE P1003.2, *Regular Expression Notation*.



Strings used with this interface may only include the ASCII characters 0x0 to 0x7F.

Basic Regular Expressions

The Basic Regular Expressions are composed of these terms:

x\$

x at end of line (\$ must be the last term).

^x

x at beginning of line (^ must be first the term).

x*

Zero or more occurrences of *x*.

.

Any single character (except newline).

c

The character *c*.

xc

x followed by the character *c*.

cx

Character *c* followed by *x*.

[cd]

The characters *c* or *d*.

[c-d]

All characters between *c* and *d*, inclusive.

[[^]*c*]

Any character but *c*.

[[:*classname*:]]

Any of the following classes:

- `alnum`
- `alpha`
- `cntrl`
- `digit`
- `graph`
- `lower`
- `print`
- `punct`
- `space`
- `upper`
- `xdigit`

[[=*c*=]]

All character in the equivalence class with *c*.

[[.=.]]

All collating elements.

***x*{*m*,*n*}**

m through *n* occurrences of *x*.

\c

Character *c*, even if *c* is an operator.

\(*x*)

A labeled subexpression, *x*.

\m

The *m*th subexpression encountered.

xy

Expression *x* followed by *y*.

Extended Regular Expressions

The Extended Regular Expressions also include:

x+

One or more occurrences of *x*.

x?

Zero or one occurrences of *x*.

(x)

Subexpression *x* (for precedence handling).

x|y

Expression *x* OR *y*.

Returns:

0

Success.

<>0

An error occurred (use [regerror\(\)](#) (p. 2547) to get an explanation).

Examples:

```
/*
   The following example prints out all lines
   from FILE "f" that match "pattern".
*/
#include <stdio.h>
#include <regex.h>
#include <limits.h>

#define BUFFER_SIZE 512

void grep( char* pattern, FILE* f )
{
    int t;
    regex_t re;
    char buffer[BUFFER_SIZE];

    if ((t=regcomp( &re, pattern, REG_NOSUB )) != 0) {
        regerror(t, &re, buffer, sizeof buffer);
        fprintf(stderr,"grep: %s (%s)\n",buffer,pattern);
        return;
    }
    while( fgets( buffer, BUFFER_SIZE, f ) != NULL ) {
        if( regexec( &re, buffer, 0, NULL, 0 ) == 0 ) {
            fputs( buffer, stdout );
        }
    }
    regfree( &re );
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Contributing author:

Henry Spencer. For license information, see the Third Party License Terms List at <http://licensing.qnx.com/third-party-terms/>.

regerror()

Explain a regular expression error code

Synopsis:

```
#include <regex.h>

size_t regerror( int err,
                 const regex_t * reg,
                 char * buf,
                 size_t len );
```

Arguments:

err

The value returned by a previous call to [regcomp\(\)](#) (p. 2542) or [regexexec\(\)](#) (p. 2549).

reg

A pointer to the `regex_t` object for the regular expression that you provided to the failed call to [regcomp\(\)](#) or [regexexec\(\)](#).

buf

A pointer to a buffer where the function can store the explanation.

len

The length of the buffer, in characters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The [regerror\(\)](#) function provides a string explaining an error code returned by [regcomp\(\)](#) or [regexexec\(\)](#). The string is copied into *buf* for up to *len* characters.

Returns:

The number of characters copied into the buffer.

Examples:

See [regcomp\(\)](#) (p. 2542).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Contributing author:

Henry Spencer. For license information, see the Third Party License Terms List at <http://licensing.qnx.com/third-party-terms/>.

regexexec()

Compare a string with a compiled regular expression

Synopsis:

```
#include <regex.h>

int regexexec( const regex_t * preg,
               const char * string,
               size_t nmatch,
               regmatch_t * pmatch,
               int eflags );
```

Arguments:***preg***

A pointer to the `regex_t` object for the regular expression that you want to execute. You must have compiled the expression by calling [`regcomp\(\)`](#) (p. 2542).

string

The string that you want to match against the regular expression.

nmatch

The maximum number of matches to record in *pmatch*.

pmatch

An array of `regmatch_t` objects where the function can record the matches; see below.

eflags

Execution parameters to `regexexec()`. For example, you may need to call `regexexec()` multiple times if the line you're processing is too large to fit into *string*. The *eflags* argument is the bitwise inclusive OR of zero or more of the following flags:

- `REG_NOTBOL` — the *string* argument doesn't point to the beginning of a line.
- `REG_NOTEOL` — the end of *string* isn't the end of a line.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `regexec()` function compares *string* against the compiled regular expression *preg*. If `regexec()` finds a match it returns zero; otherwise, it returns nonzero.

The *preg* argument represents a compiled form of either a Basic Regular Expression or Extended Regular Expression. These classes are rigorously defined in IEEE P1003.2, *Regular Expression Notation*, and are summarized in the documentation for [regcomp\(\)](#) (p. 2542).

The `regexec()` function records the matches in the *pmatch* array, with *nmatch* specifying the maximum number of matches to record. The `regmatch_t` structure is defined as:

```
typedef struct {  
    regoff_t rm_so;  
    regoff_t rm_eo;  
} regmatch_t;
```

The members are:

rm_so

The byte offset from the beginning of the string to the beginning of the matched substring.

rm_eo

One greater than the offset from the beginning of the string to the end of the matched substring.

The offsets in *pmatch[0]* identify the substring corresponding to the entire expression, while those in *pmatch[1...nmatch]* identify up to the first *nmatch* subexpressions. Unused elements of the *pmatch* array are set to -1.



You can disable the recording of substrings by either specifying `REG_NOSUB` in `regcomp()`, or by setting *nmatch* to zero.

Returns:

0

The *string* argument matches *preg*.

<>0

A match wasn't found, or an error occurred (use [regerror\(\)](#) (p. 2547) to get an explanation).

Examples:

See [regcomp\(\)](#) (p. 2542).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Contributing author:

Henry Spencer. For license information, see the Third Party License Terms List at <http://licensing.qnx.com/third-party-terms/>.

regfree()

Release memory allocated for a regular expression

Synopsis:

```
#include <regex.h>

void regfree( regex_t * preg );
```

Arguments:

preg

A pointer to the `regex_t` object for the regular expression that you want to free; see [regcomp\(\)](#) (p. 2542).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *regfree()* function releases all memory allocated by *regcomp()* associated with *preg*.

Examples:

See [regcomp\(\)](#) (p. 2542).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

Contributing author:

Henry Spencer. For license information, see the Third Party License Terms List at <http://licensing.qnx.com/third-party-terms/>.

remainder(), remainderf(), remainderl()

Compute the floating point remainder

Synopsis:

```
#include <math.h>

double remainder( double x,
                  double y );

float remainderf( float x,
                  float y );

long double remainder( long double x,
                       long double y );
```

Arguments:

x

The numerator of the division.

y

The denominator.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions return the floating point remainder $r = x - ny$, where y is nonzero. The value n is the integral value nearest the exact value x/y . When $|n - x/y| = 1/2$, the value n is chosen to be even.

The behavior of *remainder()* is independent of the rounding mode.

Returns:

The floating point remainder $r = x - ny$, where y is nonzero. If x is infinite or y is zero and the other is non-NaN, these functions return NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [*errno*](#) (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

remove()

Remove a link to a file

Synopsis:

```
#include <stdio.h>

int remove( const char * filename );
```

Arguments:

filename

The path to the file that you want to delete.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *remove()* function removes a link to a file:

- If the *filename* names a symbolic link, *remove()* removes the link, but doesn't affect the file or directory that the link goes to.
- If the *filename* isn't a symbolic link, *remove()* removes the link and decrements the link count of the file that the link refers to.

If the link count of the file becomes zero, and no process has the file open, then the space that the file occupies is freed, and no one can access the file anymore.

If one or more processes have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed.

The *remove()* function also works on directories; however, [*rmdir\(\)*](#) (p. 2658) is more efficient.

Returns:

0

The operation was successful.

Nonzero

The operation failed (*errno* (p. 539) is set).

Errors:**EACCES**

Search permission is denied for a component of *filename*, or write permission is denied on the directory containing the link to be removed.

EBUSY

The directory named by *filename* can't be unlinked because it's being used by the system or another process, and the target filesystem or resource manager considers this to be an error.

ENAMETOOLONG

The *filename* argument exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.

ENOENT

The named file doesn't exist, or *filename* is an empty string.

ENOSYS

The *remove()* function isn't implemented for the filesystem specified by *filename*.

ENOTDIR

A component of *filename* isn't a directory.

EPERM

The file named by *filename* is a directory, and either the calling process doesn't have the appropriate privileges, or the target filesystem or resource manager prohibits using *remove()* on directories.

EROFS

The directory entry to be unlinked resides on a read-only filesystem.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( remove( "vm.tmp" ) ) {
        puts( "Error removing vm.tmp!" );
        return EXIT_FAILURE;
    }
}
```

```
    }  
    return EXIT_SUCCESS;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

remque()

Remove an element from a doubly linked queue

Synopsis:

```
#include <search.h>

void remque( void *elem);
```

Arguments:

elem

A pointer to the element you want to remove.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `remque()` function removes an element from a queue made from a doubly linked list. For more information about the elements of the queue, see [insque\(\)](#) (p. 1145), which you can use to add elements.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

rename()

Rename a file

Synopsis:

```
#include <stdio.h>

int rename( const char* old,
            const char* new );
```

Arguments:

old

The path to the file that you want to rename. If the path doesn't include a directory, *rename()* looks for the file in the current working directory.

new

The new name for the file. If the path doesn't include a directory, *rename()* creates the file in the current working directory.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *rename()* function changes the name of the file indicated by *old* to the name given in *new*.

If *new* identifies an existing file or empty directory, *rename()* overwrites it.

Returns:

0

Success.

Nonzero

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

The calling program doesn't have permission to search one of the components of either path prefix, or one of the directories containing *old* or *new* denies write permission.

EBUSY

The directory named by *old* or *new* can't be renamed because another process is using it.

EEXIST

The file specified by *new* is a directory that contains files.

EINVAL

The *new* directory pathname contains the *old* directory.

EISDIR

The file specified by *new* is a directory and *old* is a file.

ELOOP

Too many levels of symbolic links.

EMLINK

The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed `LINK_MAX`.

ENAMETOOLONG

The length of *old* or *new* exceeds `PATH_MAX`.

ENOENT

The *old* file doesn't exist, or *old* or *new* is an empty string.

ENOSPC

The directory that would contain *new* can't be extended.

ENOSYS

The *rename()* function isn't implemented for the filesystem specified in *old* or *new*.

ENOTDIR

A component of either path prefix isn't a directory, or *old* is a directory and *new* isn't.

ENOTEMPTY

The file specified by *new* is a directory that contains files.

EROFS

The *rename()* would affect files on a read-only filesystem.

EXDEV

The files or directories named by *old* and *new* are on different filesystems.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( rename( "old.dat", "new.dat" ) ) {
        puts( "Error renaming old.dat to new.dat." );
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

res_init()

Initialize the Internet domain name resolver routines

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_init( void );
```

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers.

The *res_init()* routine reads the resolver configuration file (if one is present; see */etc/resolv.conf* in the *Utilities Reference*) to get the default domain name, search list, and Internet address of the local name servers. If no server is configured, the host running the resolver is tried. If not specified in the configuration file, the current domain name is defined by the hostname; the domain name can be overridden by the environment variable **LOCALDOMAIN**. Initialization normally occurs on the first call to one of the resolver routines.

Resolver configuration

Global configuration and state information used by these routines is kept in the `__res_state` structure `_res`, which is defined in `<resolv.h>`. Since most of the values have reasonable defaults, you can generally ignore them.

The `_res.options` member is a simple bit mask that contains the bitwise OR of the enabled options. The following options are defined in `<resolv.h>`:

RES_DEBUG

Print debugging messages.

RES_DEFNAMES

If this option is set, [res_search\(\)](#) (p. 2575) appends the default domain name to single-component names (those that don't contain a dot). This option is enabled by default.

RES_DNSRCH

If this option is set, [res_search\(\)](#) searches for hostnames in the current domain and in parent domains. This is used by the standard host lookup routine, [gethostbyname\(\)](#) (p. 872). This option is enabled by default.

RES_INIT

True if the initial name server address and default domain name are initialized (i.e. [res_init\(\)](#) has been called).

RES_RECURSE

Set the recursion-desired bit in queries. This is the default. Note that [res_send\(\)](#) (p. 2578) doesn't do iterative queries — it expects the name server to handle recursion.

RES_STAYOPEN

Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the mode you normally use.

RES_USEVC

Instead of UDP datagrams, use TCP connections for queries.

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

0

Success.

Nonzero

An error occurred.

Errors:

See [herror\(\)](#) (p. 1017).

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, ***LOCALDOMAIN*** contains a domain name that overrides the current domain name.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

res_mkquery()

Construct an Internet domain name query

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_mkquery( int op,
                 const char * dname,
                 int class,
                 int type,
                 const u_char * data,
                 int datalen,
                 const u_char * newrr,
                 u_char * buf,
                 int buflen );
```

Arguments:

op

Usually QUERY, but it can be also IQUERY or NS_NOTIFY_OP. Note that not all of the query types defined in <arpa/nameser.h> are supported.

dname

The domain name for the query.

class

The class of information that you want; one of:

- C_IN — ARPA Internet.
- C_CHAOS — Chaos net (MIT).
- C_HS — Hesiod name server (MIT).
- C_ANY — any class.

You typically use C_IN.

type

The type of information that you want. You typically use T_PTR, but you can use any of the T_* constants defined in <arpa/nameser.h>.

data

NULL, or a pointer to resource record data.

datalen

The length of the data.

newrr

Currently unused. This argument is intended for making update messages.

buf

A pointer to a buffer where the function can build the query.

buflen

The length of the buffer.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *res_mkquery()* function is a low-level routine that's used by [res_query\(\)](#) (p. 2569) to construct an Internet domain name query. This routine constructs a standard query message and places it in *buf*. It returns the size of the query, or -1 if the query is larger than *buflen*.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers. Global configuration and state information used by the resolver routines is kept in the structure *_res*. For more information on the options, see [res_init\(\)](#) (p. 2563).

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

The size of the prepared query, in bytes, or -1 if an error occurs.

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, **LOCALDOMAIN** contains a domain name that overrides the current domain name.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

res_query()

Query the local Internet domain name server

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query( const char * dname,
               int class,
               int type,
               u_char * answer,
               int anslen );
```

Arguments:

dname

The fully qualified domain name that you want to query.

class

The class of information that you want; one of:

- C_IN — ARPA Internet.
- C_CHAOS — Chaos net (MIT).
- C_HS — Hesiod name server (MIT).
- C_ANY — any class.

You typically use C_IN.

type

The type of information that you want. You typically use T_PTR, but you can use any of the T_* constants defined in <arpa/nameser.h>.

answer

A pointer to a buffer where the function can store the answer to the query.

anslen

The length of the buffer.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `res_query()` function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, waits for a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers. Global configuration and state information used by the resolver routines is kept in the structure `_res`. For more information on the options, see [res_init\(\)](#) (p. 2563).

The `res_query()` function uses the following lower-level routines:

- [res_mkquery\(\)](#) (p. 2566) constructs a standard query message.
- [res_send\(\)](#) (p. 2578) sends the preformatted query and returns an answer.
- [dn_comp\(\)](#) (p. 464) compresses a domain name.
- [dn_expand\(\)](#) (p. 466) expands the compressed domain name to a full domain name.

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

The length of a reply message, in bytes, or -1 if an error occurs ([h_errno](#) (p. 1012) is set).

Errors:

See [herror\(\)](#) (p. 1017).

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, **LOCALDOMAIN** contains a domain name that overrides the current domain name.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

res_querydomain()

Query the local Internet domain name server

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_querydomain( const char * name,
                    const char * domain,
                    int class,
                    int type,
                    u_char * answer,
                    int anslen );
```

Arguments:

name

The host name that you want to query.

domain

The domain name that you want to query.

class

The class of information that you want; one of:

- C_IN — ARPA Internet.
- C_CHAOS — Chaos net (MIT).
- C_HS — Hesiod name server (MIT).
- C_ANY — any class.

You typically use C_IN.

type

The type of information that you want. You typically use T_PTR, but you can use any of the T_* constants defined in <arpa/nameser.h>.

answer

A pointer to a buffer where the function can store the answer to the query.

anslen

The length of the buffer.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *res_querydomain()* function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, waits for a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the host specified by concatenating *name* and *domain*. The trailing dot is removed from *name* if *domain* is 0.

The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

Returns:

0

Success.

-1

An error occurred.

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, **LOCALDOMAIN** contains a domain name that overrides the current domain name.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	No
Thread	No

res_search()

Query a local server, using search options

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_search( const char * dname,
                int class,
                int type,
                u_char * answer,
                int anslen );
```

Arguments:

dname

The fully qualified domain name that you want to query.

class

The class of information that you want; one of:

- C_IN — ARPA Internet.
- C_CHAOS — Chaos net (MIT).
- C_HS — Hesiod name server (MIT).
- C_ANY — any class.

You typically use C_IN.

type

The type of information that you want. You typically use T_PTR, but you can use any of the T_* constants defined in <arpa/nameser.h>.

answer

A pointer to a buffer where the function can store the answer to the query.

anslen

The length of the buffer.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The `res_search()` routine makes an Internet domain name search. Like [res_query\(\)](#) (p. 2569), `res_search()` makes a query and waits for a response. But it also implements the default and search rules controlled by the `RES_DEFNAMES` and `RES_DNSRCH` options. It returns the first successful reply.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information used by the resolver routines is kept in the structure `_res`. For more information on the options, see [res_init\(\)](#) (p. 2563).

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

The length of a reply message, in bytes, or -1 if an error occurs (`h_errno` is set).

Errors:

See [herror](#) (p. 1017).

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, **LOCALDOMAIN** contains a domain name that overrides the current domain name.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	No
Thread	No

res_send()

Send a preformatted Internet domain name query

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send( const u_char * msg,
              int msglen,
              u_char * answer,
              int anslen );
```

Arguments:

msg

The preformatted Internet domain name query that you want to send.

msglen

The length of the message.

answer

A pointer to a buffer where the function can store the answer to the query.

anslen

The length of the buffer.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *res_send()* function is a low-level routine that's used by [res_query\(\)](#) (p. 2569) to send a preformatted Internet domain name query and return an answer. It calls [res_init\(\)](#) (p. 2563) if RES_INIT isn't set, sends the query to the local name server, and handles timeouts and retries.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information used by the resolver routines is kept in the structure `_res`. For more information on the options, see [res_init\(\)](#) (p. 2563).

Based on:

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035

Returns:

The length of a reply message, in bytes; or -1 if an error occurs.

Errors:

ECONNREFUSED

No name servers found.

ETIMEDOUT

No answer obtained.

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, **LOCALDOMAIN** contains a domain name that overrides the current domain name.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

resmgr_attach()

Attach a path to the pathname space

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int resmgr_attach (
    dispatch_t *dpp,
    resmgr_attr_t *attr,
    const char *path,
    enum _file_type file_type,
    unsigned flags,
    const resmgr_connect_funcs_t *connect_funcs,
    const resmgr_io_funcs_t *io_funcs,
    RESMGR_HANDLE_T *handle );
```

Arguments:

dpp

A dispatch handle created by a successful call to [dispatch_create\(\)](#) (p. 433).

attr

A pointer to a [resmgr_attr_t](#) (p. 2583) structure that defines attributes for the resource manager; see below.

path

NULL, or the path that you want to attach the resource manager to; see below.

file_type

The file type; one of the following (defined in `<sys/ftype.h>`):

- `_FTYPE_ANY` — the path name can be anything.
- `_FTYPE_LINK` — reserved for the Process Manager.
- `_FTYPE_MOUNT` — receive mount requests on the path (*path* must be NULL).
- `_FTYPE_QUEUE` — reserved for a message-queue manager.
- `_FTYPE_PIPE` — reserved for a pipe manager.
- `_FTYPE_SEM` — reserved for a semaphore manager.
- `_FTYPE_SHMEM` — reserved for a shared memory object.
- `_FTYPE_SOCKET` — reserved for a socket manager.

- `_FTYPE_SYMLINK` — reserved for the Process Manager.

flags

Flags that control the pathname resolution:

- `_RESMGR_FLAG_AFTER`
- `_RESMGR_FLAG_BEFORE` (the default)
- `_RESMGR_FLAG_DIR`
- `_RESMGR_FLAG_FTYPEALL`
- `_RESMGR_FLAG_FTYPEONLY`
- `_RESMGR_FLAG_OPAQUE`
- `_RESMGR_FLAG_SELF`

For more information, see “[The flags argument](#) (p. 2584),” below.

connect_funcs

A pointer to the [resmgr_connect_funcs_t](#) (p. 2592) structure that defines the POSIX-level connect functions.

io_funcs

A pointer to the [resmgr_io_funcs_t](#) (p. 2610) structure that defines the POSIX-level I/O functions.

handle

A pointer to an arbitrary structure that you want to associate with the pathname you're attaching. For most resource managers, this is an [iofunc_attr_t](#) (p. 1221) structure.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `resmgr_attach()` function puts the *path* into the general pathname space and binds requests on this path to the dispatch handle *dpp*.



In order to add a name to the pathname space, your process must have the `PROCmgr_AID_PATHSPACE` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Most of the above file types are used for special services that have their own open function associated with them. For example, the mqueue manager specifies *file_type* as `_FTYPE_MQUEUE`, and [mq_open\(\)](#) (p. 1709) requests a pathname match of the same type.

Specify `_FTYPE_ANY` for normal filesystems and simple devices, such as serial ports, that don't have their own special open type, or if your resource manager can handle the type of service or act as a redirection node to a manager that does. Most resource managers are of this type.

Your resource manager won't receive messages from an open of an inappropriate type. The following table shows the different open function types and the types of pathnames they'll match.

Function:	<i>file_type</i> :	Matches pathname of type:
mq_open() (p. 1709)	<code>_FTYPE_MQUEUE</code>	<code>_FTYPE_ANY</code> or <code>_FTYPE_MQUEUE</code>
open() (p. 1905)	<code>_FTYPE_ANY</code>	All types
pipe() (p. 2011)	<code>_FTYPE_PIPE</code>	<code>_FTYPE_ANY</code> or <code>_FTYPE_PIPE</code>
sem_open() (p. 2814)	<code>_FTYPE_SEM</code>	<code>_FTYPE_ANY</code> or <code>_FTYPE_SEM</code>
shm_open() (p. 2935)	<code>_FTYPE_SHMEM</code>	<code>_FTYPE_ANY</code> or <code>_FTYPE_SHMEM</code>
socket() (p. 3075)	<code>_FTYPE_SOCKET</code>	<code>_FTYPE_ANY</code> or <code>_FTYPE_SOCKET</code>

The generic [open\(\)](#) (p. 1905) can be used to open a pathname of any type.

If you want to use the POSIX functions, we've provided you with the POSIX layer; to fill your connect and I/O functions tables with the default handler functions supplied by the POSIX layer library, call [iofunc_func_init\(\)](#) (p. 1271). You can then override the defaults placed in the structures with your own handlers.



The [resmgr_attach\(\)](#) function copies the *pointers* to the `resmgr_connect_funcs_t` and `resmgr_io_funcs_t` structures, not the structures themselves. You should allocate the structures, declare them to be static, or make them global variables. If your resource manager is for more than one device with different handlers, create separate structures that define the handlers.

In the most general case, the last argument, *handle* is an arbitrary structure that you wish to have associated with the pathname you're attaching. Practically, however, we

recommend that it contain the POSIX layer's well defined attributes structure, [iofunc_attr_t](#) (p. 1221), because this lets you use the POSIX-layer default library. You can extend the data that's contained in the attributes structure to contain any device-specific data that you may require. This is commonly done, and is described in the Extending the POSIX-Layer Data Structures chapter of *Writing a Resource Manager*.

In order to use the POSIX layer default library, the attributes structure must be bound into the Open Control Block, and you must use the POSIX layer's [iofunc_ocb_t](#) (p. 1319) OCB. This is described in the documentation for [resmgr_open_bind\(\)](#) (p. 2636), as well as in the above reference.

resmgr_attr_t structure

You can specify attributes such as the maximum message size, number of parts (number of IOVs in context), and flags in the *attr* structure. The `resmgr_attr_t` structure looks like this:

```
typedef struct _resmgr_attr {
    unsigned    flags;
    unsigned    nparts_max;
    unsigned    msg_max_size;
    int         (*other_func)
                ( resmgr_context_t *, void *msg );
} resmgr_attr_t;
```

The members include:

flags

Flags that affect the behavior of the resource manager interface; 0, or a combination of the following bits (defined in `<sys/dispatch.h>`):

- **RESMGR_FLAG_ATTACH_LOCAL** — set up the resource manager, but don't register its path with `procnto`. You can send messages to the resource manager's channel (if you know where to find it).
- **RESMGR_FLAG_ATTACH_OTHERFUNC** — the *other_func* member of this structure points to a function for unhandled I/O messages.
- **RESMGR_FLAG_CROSS_ENDIAN** — the server handles cross-endian support. The framework handles all necessary conversions on the server's side; the client doesn't have to do anything.
- **RESMGR_FLAG_NO_DEFAULT_FUNC** — not implemented.
- **RESMGR_FLAG_RCM** (QNX Neutrino 6.6 or later) — automatically adopt the client's resource constraint mode when handling a request. For more information, see Resource constraint thresholds in the Processes chapter of the QNX Neutrino *Programmer's Guide*.



Don't confuse these flags with the ones you pass in the *flags* argument to *resmgr_attach()*. The names of these flags don't start with an underscore (`_`).

nparts_max

The number of components to allocate for the IOV array. If you specify 0, the resource manager library bumps the value to the minimum usable by the library itself.

msg_max_size

The minimum amount of room to reserve for receiving a message that's allocated in *resmgr_context_alloc()* (p. 2594). If the value is too low, or you specify it as 0, *resmgr_attach()* picks a value that's usable.

other_func

A pointer to a function that's called if the resource manager receives an I/O message that it didn't successfully handle. This function is attached only if `RESMGR_FLAG_ATTACH_OTHERFUNC` is set in the *flags* member of this structure.

The *flags* argument

The *flags* argument to *resmgr_attach()* specifies additional information to control the pathname resolution.



Don't confuse these flags with the ones you set in the *flags* member of the `resmgr_attr_t` structure. The names of these flags start with an underscore (`_`).

The flags (defined in `<sys/resmgr.h>`) include at least the following bits:

`_RESMGR_FLAG_AFTER`

Force the path to be resolved after others with the same pathname at the same mountpoint.

`_RESMGR_FLAG_BEFORE`

Force the path to be resolved before others with the same pathname at the same mountpoint.



If multiple registrations are made with `_RESMGR_FLAG_BEFORE` or `_RESMGR_FLAG_AFTER`, then all `BEFORE` registrations are resolved first in FIFO order, followed by the non-flagged registrations

in LIFO order, and then all AFTER registrations in LIFO order (i.e., the first BEFORE registered is resolved first, and the first AFTER registered is resolved last).

_RESMGR_FLAG_DIR

Treat the pathname as a directory and allow the resolving of longer pathnames. The `_IO_CONNECT` message contains the pathname passed to the client `open()` with the matching prefix stripped off. Without this flag, the pathname is treated as a simple file requiring an exact match.

Attached path	Opened path	<u>_RESMGR_FLAG_DIR</u> set	<u>_RESMGR_FLAG_DIR</u> clear
/a/b	/a/b	Match ""	Match ""
/a/b	/a/b/c	Match c	No match
/a/b	/a/b/c/d	Match c/d	No match
/a/b	/a/bc	No match	No match

You can't attach a directory pathname that contains, as a subset, an existing file pathname. Likewise, you can't attach a file pathname that's a subset of an existing directory pathname.

Existing path	New path	New path allowed?
Directory /a/b	Directory /a	Yes
Directory /a/b	Directory /a/b/c	Yes
File /a/b	Directory /a	Yes
File /a/b	Directory /a/b/c	No; the directory is beneath a file
Directory /a/b	File /a	No; the directory is beneath a file
Directory /a/b	File /a/b/c	Yes
File /a/b	File /a	Yes
File /a/b	File /a/b/c	Yes

_RESMGR_FLAG_FTYPEALL

Handle requests for all file types. You must specify a registration file type of `FTYPE_ALL`.

_RESMGR_FLAG_FTYPEONLY

Handle only requests for the specific filetype indicated. The pathname must be NULL.

_RESMGR_FLAG_OPAQUE

Don't resolve paths to mountpoints on a path shorter than this (i.e. find the longest match against all pathnames attached).

_RESMGR_FLAG_SELF

Allow requests to resolve back to this server; in other words, allow the resource manager to open a path to itself.



Be very careful if you set `_RESMGR_FLAG_SELF` because it's possible for a deadlock to occur. For more information, see “Robust implementations with Send/Receive/Reply” in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

Returns:

A unique link ID associated with this attach, or -1 on failure ([errno](#) (p. 539) is set).

The returned ID is needed to detach the pathname at a later time using [resmgr_detach\(\)](#) (p. 2600). The ID is also passed back in the [resmgr_handler\(\)](#) (p. 2608) function in `ctp->id`.

Errors:**EBUSY**

An internal resource wasn't available.

ENOMEM

There isn't enough free memory to complete the operation.

ENOTDIR

A component of the pathname wasn't a directory entry.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

Here's an example of a simple single-threaded resource manager:

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

int main(int argc, char **argv)
{
    dispatch_t        *dpp;
    resmgr_attr_t      resmgr_attr;
    resmgr_context_t   *ctp;
    int                id;

    /* initialize dispatch interface */
    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    /* initialize resource manager attributes */
    memset( &resmgr_attr, 0, sizeof resmgr_attr );
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* initialize functions for handling messages */
    iofunc_func_init (p. 1271)( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                               _RESMGR_IO_NFUNCS, &io_funcs );

    /* initialize attribute structure */
    iofunc_attr_init (p. 1217)( &attr, S_IFNAM | 0666, 0, 0 );

    /* attach our device name (passing in the POSIX defaults
     * from the iofunc_func_init and iofunc_attr_init functions)
     */
    if ( (id = resmgr_attach
          ( dpp, &resmgr_attr, "/dev/mynull", _FTYPE_ANY, 0,
            &connect_funcs, &io_funcs, &attr)) == -1 ) {
        fprintf( stderr, "%s: Unable to attach name.\n", \
                argv[0] );
        return EXIT_FAILURE;
    }

    /* allocate a context structure */
    ctp = resmgr_context_alloc( dpp );

    /* start the resource manager message loop */
    while (1) {
        if ( (ctp = resmgr_block( ctp )) == NULL ) {
            fprintf(stderr, "block error\n");
            return EXIT_FAILURE;
        }
        resmgr_handler(ctp);
    }
}
```

For more examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), and [thread_pool_create\(\)](#) (p. 3387). For more information on writing a resource manager, see *Writing a Resource Manager*

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_block()

Block while waiting for a message

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

resmgr_context_t * resmgr_block
    ( resmgr_context_t * ctp );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_block()* function waits for a message (created by a call to [resmgr_context_alloc\(\)](#) (p. 2594)) for context *ctp*.



This function is a special case of [dispatch_block\(\)](#) (p. 426) that you should use only with a simple resource manager. If you need to attach pulses or other messages, then you should use *dispatch_block()*.

Returns:

The same pointer as *ctp*, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender's buffers only when *MsgReceive()* is called, a fault could occur *in the sender* if the sender's buffers are invalid.

If a fault occurs when accessing the sender buffers (only) they'll receive an EFAULT and the *MsgReceive()* won't unblock.

EINTR

The call was interrupted by a signal.

ETIMEDOUT

A kernel timeout (that was set with *dispatch_timeout()* (p. 443)) unblocked the call.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    resmgr_context_t *ctp;

    if ((dpp = dispatch_create()) == NULL) {
        fprintf( stderr, "%s: Unable to allocate \
            dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }

    ...

    ctp = resmgr_context_alloc( dpp );

    while (1) {
        if ( ( ctp = resmgr_block( ctp ) ) == NULL ) {
            fprintf( stderr, "block error\n" );
            return EXIT_FAILURE;
        }
        resmgr_handler( ctp );
    }
}
```

For examples using the dispatch interface, see *dispatch_create()* (p. 433), *message_attach()* (p. 1624), *resmgr_attach()* (p. 2580), and *thread_pool_create()* (p. 3387).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Use *resmgr_block()* only in a simple resource manager and when you don't use [message_attach\(\)](#) (p. 1624), [pulse_attach\(\)](#) (p. 2442), or [select_attach\(\)](#) (p. 2796).

resmgr_connect_funcs_t

Table of POSIX-level connect functions

Synopsis:

```
#include <sys/resmgr.h>

typedef struct _resmgr_connect_funcs {
    unsigned nfuncs;

    int (*open)      (resmgr_context_t *ctp, io_open_t *msg,
                     RESMGR_HANDLE_T *handle, void *extra);

    int (*unlink)    (resmgr_context_t *ctp, io_unlink_t *msg,
                     RESMGR_HANDLE_T *handle, void *reserved);

    int (*rename)    (resmgr_context_t *ctp, io_rename_t *msg,
                     RESMGR_HANDLE_T *handle,
                     io_rename_extra_t *extra);

    int (*mknod)     (resmgr_context_t *ctp, io_mknod_t *msg,
                     RESMGR_HANDLE_T *handle, void *reserved);

    int (*readlink)  (resmgr_context_t *ctp, io_readlink_t
                     *msg,
                     RESMGR_HANDLE_T *handle, void *reserved);

    int (*link)      (resmgr_context_t *ctp, io_link_t *msg,
                     RESMGR_HANDLE_T *handle,
                     io_link_extra_t *extra);

    int (*unblock)   (resmgr_context_t *ctp, io_pulse_t *msg,
                     RESMGR_HANDLE_T *handle, void *reserved);

    int (*mount)     (resmgr_context_t *ctp, io_mount_t *msg,
                     RESMGR_HANDLE_T *handle,
                     io_mount_extra_t *extra);
} resmgr_connect_funcs_t;
```

Description:

The `resmgr_connect_funcs_t` structure is a table of the POSIX-level connect functions that are used by a resource manager. You can initialize this table by calling [*iofunc_func_init\(\)*](#) (p. 1271) and then overriding the defaults with your own functions.

This structure includes *nfuncs*, which indicates how many functions are in the table (in case the structure grows in the future), along with these functions:

Member:	Used to:	Default:
<i>open</i>	Handle <code>_IO_CONNECT</code> messages	iofunc_open_default() (p. 1325)
<i>unlink</i>	Unlink the resource	None
<i>rename</i>	Rename the resource	None
<i>mknod</i>	Create a filesystem entry point	None
<i>readlink</i>	Read a symbolic link	None
<i>link</i>	Create a symbolic link	None
<i>unblock</i>	Unblock the resource if an operation is aborted	None
<i>mount</i>	Mount a filesystem	None

Classification:

[QNX Neutrino](#) (p. 46)

resmgr_context_alloc()

Allocate a resource-manager context

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

resmgr_context_t * resmgr_context_alloc
( dispatch_t * dpp );
```

Arguments:

dpp

A dispatch handle created by a successful call to [dispatch_create\(\)](#) (p. 433).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_context_alloc()* function returns a context that's used for [blocking](#) (p. 2589) and receiving messages.



This function is a special case of [dispatch_context_alloc\(\)](#) (p. 429). You should use it only when writing a simple resource manager.

Returns:

A pointer to a [resmgr_context_t](#) (p. 2598) structure, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

No resource manager events were attached to *dpp*.

ENOMEM

Insufficient memory to allocate the context *ctp*.

Examples:

```

#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    resmgr_context_t *ctp;

    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
    ...

    if ( (ctp = resmgr_context_alloc( dpp )) == NULL ) {
        fprintf( stderr, "Context wasn't allocated.\n" );
        return EXIT_FAILURE;
    }
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_context_free()

Free a resource-manager context

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void resmgr_context_free( resmgr_context_t * ctp );
```

Arguments:

ctp

A pointer to the [resmgr_context_t](#) (p. 2598) structure that you want to free.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_context_free()* function frees a context allocated by [resmgr_context_alloc\(\)](#) (p. 2594).



This function is a special case of [dispatch_context_free\(\)](#) (p. 431). You should use it only when writing a simple resource manager.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    resmgr_context_t *ctp;

    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                    dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
    ...

    if ( (ctp = resmgr_context_alloc( dpp )) == NULL ) {
        fprintf( stderr, "Context wasn't allocated.\n" );
    }
}
```



```

    }
    return EXIT_FAILURE;
}
...
resmgr_context_free ( ctp );
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_context_t

Context information that's passed between resource-manager functions

Synopsis:

```
#include <sys/resmgr.h>

typedef struct _resmgr_context {
    int          rcvid;
    struct _msg_info info;
    resmgr_iomsgs_t *msg;
    dispatch_t   *dpp;
    int          id;
    unsigned     tid;
    unsigned     msg_max_size;
    int          status;
    int          offset;
    int          size;
    iov_t        iov[1];
} resmgr_context_t;
```

Description:

The `resmgr_context_t` structure defines context information that's passed to resource-manager functions.

The members include:

rcvid

The receive ID to use for messages to and from the client.

info

A [_msg_info](#) (p. 1731) structure that contains information about the message received by the resource manager.

msg

A pointer to the message received by the resource manager, expressed as a union of all the possible message types.

dpp

The dispatch handle, created by a successful call to [dispatch_create\(\)](#) (p. 433).

id

The link Id, returned by [resmgr_attach\(\)](#) (p. 2580).

tid

Not used; always zero.

msg_max_size

The minimum amount of space reserved for receiving a message.

status

A place to store the status of the current operation. Always use [_RESMGR_STATUS\(\)](#) (p. 2643) to set this member.

offset

The offset, in bytes, into the client's message. You'll use this when working with combine messages.

size

The number of valid bytes in the message area.

iov

An I/O vector where you can place the data that you're returning to the client.

Classification:

[QNX Neutrino](#) (p. 46)

resmgr_detach()

Remove a pathname from the pathname space

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int resmgr_detach( dispatch_t * dpp,
                  int id,
                  unsigned flags );
```

Arguments:

dpp

A dispatch handle created by a successful call to [dispatch_create\(\)](#) (p. 433).

id

The link ID that [resmgr_attach\(\)](#) (p. 2580) returned.

flags

Flags that affect the operation. The possible *flags* (defined in `<sys/dispatch.h>` and `<sys/resmgr.h>`) are:

- `_RESMGR_DETACH_ALL` — detach the name from the namespace and invalidate all open bindings.
- `_RESMGR_DETACH_CLOSE` — close all bindings when detaching.
- `_RESMGR_DETACH_PATHNAME` — detach only the name from the namespace, leaving existing bindings intact. This option is useful when you're unlinking a file or device, and you want to remove the name, but you want processes with open files to continue to use it until they close.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_detach()* function removes pathname *id* from the pathname space of context *dpp*.

Blocking states

The *resmgr_detach()* function blocks until the `RESMGR_HANDLE_T`, that's passed to the corresponding *resmgr_attach()* (p. 2580), isn't being used in any connection function.

The effect that this has on servers is generally minimal. You should follow the following precautions to prevent potential deadlock situations:

- If you're using the `RESMGR_HANDLE_T` as an attribute, and that attribute is locked in any of the connection callouts (i.e. open, unlink, mount, etc.), then should unlock it before calling *resmgr_detach()*. This allows any pending connection requests to complete before they're consequently invalidated.



If you call *resmgr_detach()* from within a connection function, then the internal reference counting takes this into account and the server doesn't deadlock.

- If two or more *resmgr_detach()* requests come in simultaneously, only one of the requests is served. The superfluous request will return with an error of -1 and *errno* set to `ENOENT` to indicate that the detachment process has already begun, and the entry is now invalid. If dynamically allocated, you should release `RESMGR_HANDLE_T` only after a successful return from *resmgr_detach()*.
- If *resmgr_detach()* is called and an existing client connection is established, then the I/O callout table is redirected for that client connection. The client will receive an error of `EBADF` when it uses the *fd* associated with that connection.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *id* was never attached with *resmgr_attach()* (p. 2580).

ENOENT

A previous detachment request is in progress, or the *id* has already been detached.

Examples:

```

#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int id;

    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
            dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    id = resmgr_attach ( ... );

    ...

    if ( resmgr_detach( dpp, id, 0 ) == -1 ) {
        fprintf( stderr, "Failed to remove pathname \
            from the pathname space.\n" );
        return EXIT_FAILURE;
    }
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_devino()

Get the device and inode number

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_devino( int id,
                  dev_t *pdevno,
                  ino64_t *pino );
```

Arguments:

id

The link ID that [resmgr_attach\(\)](#) (p. 2580) returned.

pdevno

A pointer to a `dev_t` object where the function can store the device number.

pino

A pointer to a `ino64_t` object where the function can store the inode number.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `resmgr_devino()` fills in the structures pointed to by *pdevno* and *pino* with the device number and inode number extracted from *id*.

This function is typically used to fill in:

- `iofunc_mount_t->dev`
- `iofunc_attr_t->inode`

You can use the `major()`, `minor()`, and `makedev()` macros to work with device IDs. They're defined in `<sys/types.h>` and are described in the documentation for [stat\(\)](#) (p. 3164).

Returns:

-1 on error ([errno](#) (p. 539) is set); any other value on success.

Errors:

EINVAL

The *id* argument is invalid.

Examples:

```
#include <sys/resmgr.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    iofunc_mount_t    mount;
    iofunc_attr_t     attr;

    ...
    attr.mount = &mount;
    ...
    id = resmgr\_attach (p. 2580)( ... )
    ...
    resmgr_devino(id, &mount.dev, &attr.inode);
    ...
    return EXIT_SUCCESS;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_handle_grow()

Expand the capacity of the device manager database

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_handle_grow ( unsigned min );
```

Arguments:

min

The number of requests that you want to accommodate.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *resmgr_handle_grow()* function pre-grows or allocates the resource manager database table entries to support a given number of connections to improve runtime performance by reducing the number of dynamic memory allocations.

The function pre-allocates database space for *min* requests.

Returns:

The number of free entries in the table, or -1 if the resource manager table can't be locked.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_handle_tune()

Tune aspects of client fd-to-OCB mapping

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_handle_tune(int min_handles,
                      int min_clients,
                      int max_client_handles,
                      int *min_handles_old,
                      int *min_clients_old,
                      int *max_client_handles_old);
```

Arguments:

min_handles

To perform the described mapping, the resource manager framework makes use of `_resmgr_handle_entry` structures. This value describes the minimum number of these structures to keep around. If more than this number are in use, they may be returned to the heap via `free()` as they're released.

min_clients, max_client_handles

To perform the described mapping, the resource manager framework makes use of hash buckets, one per client. The *min_clients* describes the minimum number of these buckets to keep around. If more than this number of clients are in communication with your resource manager, these buckets may be released back to the heap via `free()` as particular clients close all their *fds* to your manager.

The *max_client_handles* describes the size of each of these hash buckets. The maximum number of lookups to find a particular *fd*-to-OCB mapping is the client's max *fd* divided by *max_client_handles* rounded to the nearest integer, i.e. in pseudocode:

`ceil(max fd/max_client_handles).`

If this value changes, the new value takes effect for newly connected clients. Existing clients are unaffected.

If negative values are specified to any of the above three parameters, their current values are left unchanged.

****_old***

If any of these are non-NULL, the corresponding value in use by the resource manager layer at the time of the call is returned.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

One of the functions of the resource manager framework is to perform the mapping of client file descriptors to structures local to the resource manager that describe these descriptors. These structures are often Open Control Blocks (OCBs). For details on OCBs, see [resmgr_open_bind\(\)](#) (p. 2636). You can use `resmgr_handle_tune()` to tune certain aspects of this mapping and subsequent lookups of a client's OCBs.

Returns:

0

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_handler()

Handle resource manager messages

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int resmgr_handler( resmgr_context_t * ctp );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_handler()* function handles the message received in *ctp*. This function handles different I/O messages through the resource manager framework.



The *resmgr_handler()* function is a special case of [dispatch_handler\(\)](#) (p. 440). You should use it only when writing a simple resource manager i.e. where there's no need to attach pulses or messages.

Returns:

0

Success.

-1

An error occurred.

Examples:

```
#include <sys/dispatch.h>
```

```

#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    resmgr_context_t *ctp;

    if ((dpp = dispatch_create()) == NULL) {
        fprintf( stderr, "%s: Unable to allocate \
            dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }
    ...

    ctp = resmgr_context_alloc( dpp );

    while (1) {
        if ( ( ctp = resmgr_block( ctp ) ) == NULL ) {
            fprintf( stderr, "block error\n" );
            return EXIT_FAILURE;
        }
        resmgr_handler( ctp );
    }
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If you attach messages or pulses to *dpp* by calling [message_attach\(\)](#) (p. 1624), [pulse_attach\(\)](#) (p. 2442), or [select_attach\(\)](#) (p. 2796), those events aren't dispatched by *resmgr_handler()*. Instead, you should call [dispatch_handler\(\)](#) (p. 440).

resmgr_io_funcs_t

Table of POSIX-level I/O functions

Synopsis:

```
#include <sys/resmgr.h>

typedef struct _resmgr_io_funcs {
    unsigned nfuncs;
    int (*read)      (resmgr_context_t *ctp, io_read_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*write)     (resmgr_context_t *ctp, io_write_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*close_ocb) (resmgr_context_t *ctp, void *reserved,
                     RESMGR_OCB_T *ocb);
    int (*stat)      (resmgr_context_t *ctp, io_stat_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*notify)     (resmgr_context_t *ctp, io_notify_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*devctl)     (resmgr_context_t *ctp, io_devctl_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*unblock)    (resmgr_context_t *ctp, io_pulse_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*pathconf)   (resmgr_context_t *ctp, io_pathconf_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*lseek)      (resmgr_context_t *ctp, io_lseek_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*chmod)      (resmgr_context_t *ctp, io_chmod_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*chown)      (resmgr_context_t *ctp, io_chown_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*utime)      (resmgr_context_t *ctp, io_utime_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*openfd)     (resmgr_context_t *ctp, io_openfd_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*fdinfo)     (resmgr_context_t *ctp, io_fdinfo_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*lock)       (resmgr_context_t *ctp, io_lock_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*space)      (resmgr_context_t *ctp, io_space_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*shutdown)   (resmgr_context_t *ctp, io_shutdown_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*mmap)       (resmgr_context_t *ctp, io_mmap_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*msg)        (resmgr_context_t *ctp, io_msg_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*reserved)   (resmgr_context_t *ctp, void *msg,
                     RESMGR_OCB_T *ocb);
    int (*dup)        (resmgr_context_t *ctp, io_dup_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*close_dup)  (resmgr_context_t *ctp, io_close_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*lock_ocb)   (resmgr_context_t *ctp, void *reserved,
                     RESMGR_OCB_T *ocb);
};
```

```

    int (*unlock_ocb)(resmgr_context_t *ctp, void *reserved,
                     RESMGR_OCB_T *ocb);
    int (*sync)      (resmgr_context_t *ctp, io_sync_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*power)     (resmgr_context_t *ctp, io_power_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*acl)       (resmgr_context_t *ctp, io_acl_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*pause)     (resmgr_context_t *ctp, void *reserved,
                     RESMGR_OCB_T *ocb);
    int (*unpause)   (resmgr_context_t *ctp, io_pulse_t *msg,
                     RESMGR_OCB_T *ocb);
} resmgr_io_funcs_t;

```

Description:

The `resmgr_connect_funcs_t` structure is a table of the POSIX-level I/O functions that are used by a resource manager. You can initialize this table by calling [iofunc_func_init\(\)](#) (p. 1271) and then overriding the defaults with your own functions.

This structure includes *nfuncs*, which indicates how many functions are in the table (in case the structure grows in the future), along with these functions:

Member:	Used to:	Default:
<i>read</i>	Handle <code>_IO_READ</code> messages	iofunc_read_default() (p. 1337)
<i>write</i>	Handle <code>_IO_WRITE</code> messages	iofunc_write_default() (p. 1380)
<i>close_ocb</i>	Return the memory allocated for an OCB	iofunc_close_ocb_default() (p. 1255)
<i>stat</i>	Handle <code>_IO_STAT</code> messages	iofunc_stat_default() (p. 1355)
<i>notify</i>	Handle <code>_IO_NOTIFY</code> messages	None
<i>devctl</i>	Handle <code>_IO_DEVCTL</code> messages	iofunc_devctl_default() (p. 1261)
<i>unblock</i>	Unblock the resource if an operation is aborted	iofunc_unblock_default() (p. 1368)
<i>pathconf</i>	Handle <code>_IO_PATHCONF</code> messages	iofunc_pathconf_default() (p. 1335)
<i>lseek</i>	Handle <code>_IO_LSEEK</code> messages	iofunc_lseek_default() (p. 1289)
<i>chmod</i>	Handle <code>_IO_CHMOD</code> messages	iofunc_chmod_default() (p. 1237)

Member:	Used to:	Default:
<i>chown</i>	Handle <code>_IO_CHOWN</code> messages	<i>iofunc_chown_default()</i> (p. 1242)
<i>utime</i>	Handle <code>_IO_UTIME</code> messages	<i>iofunc_utime_default()</i> (p. 1378)
<i>openfd</i>	Handle <code>_IO_OPENFD</code> messages	<i>iofunc_openfd_default()</i> (p. 1330)
<i>fdinfo</i>	Handle <code>_IO_FDINFO</code> messages	<i>iofunc_fdinfo_default()</i> (p. 1268)
<i>lock</i>	Handle <code>_IO_LOCK</code> messages	<i>iofunc_lock_default()</i> (p. 1281)
<i>space</i>	Allocate or free memory for the resource.	None
<i>shutdown</i>	Reserved	None
<i>mmap</i>	Handle <code>_IO_MMAP</code> messages	<i>iofunc_mmap_default()</i> (p. 1298)
<i>msg</i>	Handle a more general messaging scheme to control the device	None
<i>reserved</i>	Not applicable	None
<i>dup</i>	Handle <code>_IO_DUP</code> messages	None — handled by the base layer
<i>close_dup</i>	Handle <code>_IO_CLOSE</code> messages	<i>iofunc_close_dup_default()</i> (p. 1251)
<i>lock_ocb</i>	Lock the attributes for a group of messages	<i>iofunc_lock_ocb_default()</i> (p. 1284)
<i>unlock_ocb</i>	Unlock the attributes for a group of messages	<i>iofunc_unlock_ocb_default()</i> (p. 1373)
<i>sync</i>	Handle <code>_IO_SYNC</code> messages	<i>iofunc_sync_default()</i> (p. 1360)
<i>power</i>		
<i>acl</i>	Handle <code>_IO_ACL</code> (access control list) messages	<i>iofunc_acl_default()</i> (p. 1215)
<i>pause</i>		

Member:	Used to:	Default:
<i>unpause</i>		



If you use [iofunc_lock_default\(\)](#) (p. 1281), you must also use [iofunc_close_dup_default\(\)](#) (p. 1251) and [iofunc_unblock_default\(\)](#) (p. 1368).

Classification:

[QNX Neutrino](#) (p. 46)

resmgr_iofuncs()

Extract the I/O function pointers associated with client connections

Synopsis:

```
#include <sys/resmgr.h>

const resmgr_io_funcs_t * resmgr_iofuncs(
    resmgr_context_t * ctp);
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_iofuncs()* function retrieves the I/O function callout table associated with the client connections described by the context information pointed to by *ctp*.

Returns:

A pointer to the [resmgr_io_funcs_t](#) (p. 2610) I/O function callout table, or NULL if an error occurred.

Errors:

ESRCH

The connection can't be located in the resource manager's table.

ENOMEM

There is no memory available for the operation.

EINVAL

Invalid arguments were used.

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_msg_again()

Process a message again in a resource manager

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msg_again( resmgr_context_t *ctp,
                     int rcvid);
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

rcvid

The receive ID of the message that you want to process again.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_msg_again()* function reprocesses the message corresponding to the given *rcvid*. It does this by:

- calling [MsgInfo\(\)](#) (p. 1745) and [MsgRead\(\)](#) (p. 1754) to refresh the *resmgr_context_t* structure to be as it was as when you originally received the message
- calling [MsgCurrent\(\)](#) (p. 1734) to adjust your server threads' priority to that of the blocked client
- processing the message as if it had just arrived

You use this function if your resource manager can't process a message right away. For example, if a read operation is blocked, your `io_read` handler would save the *rcvid* of the request, and then return `_RESMGR_NOREPLY`, leaving the client blocked. Later, when the resource manager can handle the request, it can call *resmgr_msg_again()* to reprocess the message.

Returns:

-1

Failure.

0

Success.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

ESRCH

The thread indicated by *rcvid* doesn't exist, has had its connection detached, or isn't in either *STATE_REPLY* or *STATE_NET_REPLY*, or it isn't blocked on the connection associated with the *rcvid*.

ESRVFAULT

The receive side of a message transfer encountered a memory fault accessing the receive/reply buffer.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_msgread()

Read a message from a client

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgread( resmgr_context_t * ctp,
                    void * msg,
                    int size,
                    int offset );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.

msg

A pointer to a buffer where the function can store the data.

bytes

The number of bytes that you want to read. These functions don't let you read past the end of the thread's message; they return the number of bytes actually read.

offset

An offset into the thread's send message that indicates where you want to start reading the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_msgread()* function is a convenience function that you should in a resource manager instead of [MsgRead\(\)](#) (p. 1754).

You'll use *resmgr_msgread()* when you handle *combine messages*, where the offset of the rest of the message that's to be read is additionally offset by previous combine message elements. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

The same values as *MsgRead()* (p. 1754): the number of bytes read, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EFAULT

A fault occurred in a server's address space when it tried to access the caller's message buffers.

ESRCH

The thread indicated by *ctp* -> *rcvid* doesn't exist or its connection is detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

resmgr_msgreadv()

Read a message from a client

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgreadv( resmgr_context_t * ctp,
                    iov_t * rmsg,
                    int rparts,
                    int offset );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.

riov

An array of buffers where the functions can store the data.

rparts

The number of elements in the *riov* array.

offset

An offset into the thread's send message that indicates where you want to start reading the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This *resmgr_msgreadv()* function is a convenience function that you should use in a resource manager instead of [MsgReadv\(\)](#) (p. 1757).

You'll use *resmgr_msgreadv()* when handling combine messages, where the offset of the rest of the message that is to be read is additionally offset by previous combine

message elements. For more information, see “Layers in a resource manager” in the Bones of a Resource Manager chapter of *Writing a Resource Manager*.

Returns:

The same values as [MsgReadv\(\)](#) (p. 1757): the number of bytes read, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EFAULT

A fault occurred in a server's address space when it tried to access the caller's message buffers.

ESRCH

The thread indicated by *ctp* -> *rcvid* doesn't exist or has had its connection detached.

ESRVRFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

resmgr_msgreply()

Reply to client with a message

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgreply( resmgr_context_t *ctp,
                    void *ptr,
                    int len );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions. This function extracts the *rcvid* from this structure.

len

The number of bytes that you want to write. This function doesn't let you write past the end of the sender's buffer; it returns the number of bytes actually written.

ptr

A pointer that contains the message that you want to reply with.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The purpose of using *resmgr_msgreply()*, instead of using *MsgReply()* is that *resmgr_msgreply()* supports endian-swapping of standard *resmgr* message types, whereas *MsgReply()* doesn't. The function *resmgr_msgreply()* is a cover for [msgreply\(\)](#) (p. 1775) and performs the exact same functionality. The *resmgr_msgreply()* kernel call replies with a message to the thread identified by *ctp->rcvid*. This function is called by the server to reply back to the client. The client thread must already be in state `REPLY`. The thread being replied to must be in the `REPLY-blocked` state. Any thread

in the receiving process is free to reply to the message; however, it may be replied to only once for each receive.



You can fill the client's reply buffer as data becomes available by using *resmgr_msgwrite()* and *resmgr_msgwritev()*; however, you'll need to use the *resmgr_msgpepely()* or *resmgr_msgpeplyv()* at some point to unblock the client.

The function whose name contains the “v” suffix is the multipart message version for the function.

The data is taken from the array of message buffers pointed to by *ptr*. The number of elements in this array is given by *len*. The size of the message is the sum of the sizes of each buffer.

The number of bytes transferred is the minimum of that specified by both the replier and the sender. The reply data isn't allowed to overflow the reply buffer area provided by the sender.

The data transfer occurs immediately, and the replying thread doesn't block. There's no need to reply to received messages in any particular order, but you must eventually reply to each message to allow the sending thread(s) to continue execution.

Blocking states

None. In the network case, lower priority threads may run.

Returns:

If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

Errors:

EFAULT

A fault occurred in the sender's address space when a server tried to access the sender's return message buffers.

ESRCH

The thread indicated by *rcvid* doesn't exist, or is no longer REPLY-blocked on the channel, or the connection is detached.

ESRVRFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

resmgr_msgreplyv()

Reply to client with a message

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgreplyv( resmgr_context_t *ctp,
                     struct iovec *iov,
                     int parts );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions. This function extracts the *rcvid* from this structure.

iov

An array of buffers that contains the message that you want to reply with.

parts

The number of elements in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The purpose of using *resmgr_msgreplyv()*, instead of using *MsgReply()* is that *resmgr_msgreplyv()* supports endian-swapping of standard *resmgr* message types, whereas *MsgReply()* doesn't. The function *resmgr_msgreplyv()* is a cover for [msgreplyv\(\)](#) (p. 1778) and performs the exact same functionality. The *resmgr_msgreplyv()* kernel call replies with a message to the thread identified by *ctp->rcvid*. This function is called by the server to reply back to the client. The client thread must already be in state `REPLY`. The thread being replied to must be in the `REPLY-blocked` state. Any thread in the receiving process is free to reply to the message; however, it may be replied to only once for each receive.



You can fill the client's reply buffer as data becomes available by using *resmgr_msgwrite()* and *resmgr_msgwritev()*; however, you'll need to use the *resmgr_msgreply()* or *resmgr_msgreplyv()* at some point to unblock the client.

The function whose name contains the “v” suffix is the multipart message version for the function.

The data is taken from the array of message buffers pointed to by *iov*. The number of elements in this array is given by *parts*. The size of the message is the sum of the sizes of each buffer.

The number of bytes transferred is the minimum of that specified by both the replier and the sender. The reply data isn't allowed to overflow the reply buffer area provided by the sender.

The data transfer occurs immediately, and the replying thread doesn't block. There's no need to reply to received messages in any particular order, but you must eventually reply to each message to allow the sending thread(s) to continue execution.

It's quite common to reply with two-part messages consisting of a fixed header and a buffer of data. The *resmgr_msgreplyv()* function gathers the data from the buffer list into a logical contiguous message and transfers it to the sender's reply buffer(s). The sender doesn't need to specify the same number or size of buffers. The data is laid down filling each buffer as required. The filesystem, for example, builds a reply list pointing into its cache in order to reply with what appears to be one contiguous piece of data.

Blocking states

None. In the network case, lower priority threads may run.

Returns:

If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

Errors:

EFAULT

A fault occurred in the sender's address space when a server tried to access the sender's return message buffers.

ESRCH

The thread indicated by *rcvid* doesn't exist, or is no longer REPLY-blocked on the channel, or the connection is detached.

ESRVRFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

resmgr_msgwrite()

Write a message to a client

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgwrite( resmgr_context_t *ctp,
                    const void *msg,
                    int size,
                    int offset );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.

msg

A pointer to a buffer that contains the data you want to write.

size

The number of bytes that you want to write. This function doesn't let you write past the end of the sender's buffer; it returns the number of bytes actually written.

offset

An offset into the sender's buffer that indicates where you want to start writing the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `resmgr_msgwrite()` is a cover for [MsgWrite\(\)](#) (p. 1819) and performs the exact same functionality.

Returns:

The same values as *MsgWrite()*; the number of bytes written, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EFAULT**

A fault occurred in the sender's address space when a server tried to access the sender's return message buffer.

ESRCH

The thread indicated by *ctp* -> *rcvid* does not exist or has had its connection detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

resmgr_msgwritev()

Write a message to a client

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgwritev( resmgr_context_t *ctp,
                     const iov_t *smsg,
                     int sparts,
                     int offset );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.

iov

An array of buffers that contains the data you want to write.

parts

The number of elements in the array. This function doesn't let you write past the end of the sender's buffer; it returns the number of bytes actually written.

offset

An offset into the sender's buffer that indicates where you want to start writing the data.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_msgwritev()* function is a cover function for [MsgWritev\(\)](#) (p. 1822), and performs the exact same functionality. It's provided for consistency with [resmgr_msgwrite\(\)](#) (p. 2628).

Returns:

The number of bytes written, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EFAULT**

A fault occurred in the sender's address space when a server tried to access the sender's return message buffer.

ESRCH

The thread indicated by *ctp* -> *rcvid* does not exist or has had its connection detached.

ESRVRFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_RESMGR_NPARTS()

Indicate how many parts from the `ctp->iov` structure to return to the client

Synopsis:

```
#include <sys/resmgr.h>

#define _RESMGR_NPARTS( int num ) ...
```

Arguments:

num

The number of parts that you want the resource manager library to return to the client.

Library:

None.

Description:

The macro `_RESMGR_NPARTS()` indicates to the resource manager library that it should return *num* parts from the `ctp->iov` structure (see [resmgr_context_t](#) (p. 2598)) to the client. In effect, the library does the following:

[MsgReply](#) (p. 1775)(`ctp->rcvid`, `ctp->status`, `ctp->iov`, *num*).

Returns:

The negative of *num* (to distinguish the number from error codes that the handler functions might return).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

The [resmgr_attach\(\)](#) (p. 2580) function should set `attr->nparts_max` to be the maximum value for `num`.

resmgr_ocb()

Retrieve an Open Control Block

Synopsis:

```
#include <sys/resmgr.h>

void * resmgr_ocb( resmgr_context_t * ctp);
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_ocb()* function queries the internal resource manager database, which maps client connections to the server Open Control Block (OCB), to retrieve the OCB pointer that was previously bound using [resmgr_open_bind\(\)](#) (p. 2636).

Returns:

A pointer to the OCB for the matching binding, or NULL if the binding can't be found or an error occurred.

The OCB can be a structure that you define. By default, it's of type [iofunc_ocb_t](#) (p. 1319).

Errors:

ESRCH

The connection can't be located in the resource manager table.

ENOMEM

There isn't enough memory available for the operation.

EINVAL

Invalid arguments were used.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_open_bind()

Associate an OCB with an open request

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_open_bind(
    resmgr_context_t* ctp,
    void* ocb,
    const resmgr_io_funcs_t* iofuncs );
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

ocb

A pointer to the Open Control Block that you want to bind to the open request. An OCB is usually a structure of type [iofunc_ocb_t](#) (p. 1319), but you can define your own.

iofuncs

A pointer to the [resmgr_io_funcs_t](#) (p. 2610) structure that defines the I/O functions for the resource manager.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_open_bind()* function is the lowest-level call in the resource manager library used for handling open messages. It associates the Open Control Block (OCB) with a process identified by the *id* and *info* members of *ctp*.



You must use this function as part of the handling of an `_IO_OPEN` message. In practice, you don't call this function directly; you typically use either [iofunc_open_default\(\)](#) (p. 1325) or [iofunc_ocb_attach\(\)](#) (p. 1311). (The

iofunc_open_default() function calls *iofunc_ocb_attach()*, which in turn calls *resmgr_open_bind()*.

An internal data structure is allocated that maintains the number of links to the OCB. On a file descriptor *dup()*, the link count is incremented and on a *close()* it's decremented. When the count reaches zero, the *close_ocb()* callout specified in *io_funcs* is called.

In the most general case, the OCB is an arbitrary structure that *you* define that can hold information describing an open file, or just a simple `int` to hold the open mode for checking in the *read()* and *write()* callouts.

In the typical case, however, the OCB is a structure that contains at least the members as defined by the typedef `iofunc_ocb_t` (p. 1319). This typedef defines a common OCB structure that can then be used by the POSIX layer helper functions (all functions beginning with the name *iofunc_**). The advantage of this approach is that your resource manager gets POSIX behavior for free, without any additional work on your part.

The *attr* argument to the *open()* callout is also typically saved in the OCB. The well defined `iofunc_ocb_t` has a member called *attr* to which you must assign the value of the *attr* argument. This lets the POSIX helper functions access information about the current open session (as stored in the OCB) as well as information about the device itself (as stored in the attributes structure, *ocb -> attr*).

For a detailed discussion, including several examples, see *Writing a Resource Manager*.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *id* and/or *info* members of *ctp* aren't valid.

ENOMEM

Insufficient memory to allocate an internal data structure.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

resmgr_pathname()

Return the pathname associated with an ID

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_pathname( int id,
                    unsigned flags,
                    char* path,
                    int maxbuf );
```

Arguments:

id

The link ID that [resmgr_attach\(\)](#) (p. 2580) returned.

flags

Flags that affect the operation:

- `_RESMGR_PATHNAME_LOCALPATH` — get a shortened pathname that's usable only on the local machine. By default, this function gets a globally unique pathname.

path

A pointer to a buffer where the function can store the path name.

maxbuf

The size of the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_pathname()* function returns the pathname associated with an *id* that's returned from *resmgr_attach()*, it's also the *ctp->id* value of all the resmgr callout functions.

If the *id* was obtained from calling *resmgr_attach()* with `_RESMGR_FLAG_DIR` specified, then the path name includes a trailing slash.

By default, this function calls:

```
netmgr_ndtostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)
```

If you specify `_RESMGR_PATHNAME_LOCALPATH`, it calls

```
netmgr_ndtostr(ND2S_DIR_SHOW|ND2S_LOCAL_STR, nd, buf, sizeofbuf)
```

to return a shortened path that's usable on your local node only. This is useful for display.

Returns:

The length of the path, including the terminating NULL character, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EFAULT

A fault occurred in a server's address space when it tried to access the caller's message buffers.

ESRCH

The thread indicated by *ctp* -> *rcvid* doesn't exist or its connection is detached.

ESRVFAULT

A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_RESMGR_PTR()

Get one part from the `ctp->iov` structure and fill in its fields

Synopsis:

```
#include <sys/resmgr.h>

#define _RESMGR_PTR( resmgr_context_t ctp,
                    void msg,
                    size_t nbytes ) ...
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

msg

The value you want to use for the structure's *iov_base* member.

nbytes

The value you want to use for the structure's *iov_len* member.

Library:

None.

Description:

The `_RESMGR_PTR()` macro gets one part from the `ctp->iov` structure (see [resmgr_context_t](#) (p. 2598)) and fills in its fields. The macro is equivalent to:

[SETIOV](#) (p. 2870) (`ctp->iov, msg, nbytes`)
returning [_RESMGR_NPARTS \(1\)](#) (p. 2632).

Returns:

-1

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

_RESMGR_STATUS()

Set the status member of a resource-manager context

Synopsis:

```
#include <sys/resmgr.h>

#define _RESMGR_STATUS( resmgr_context_t *ctp,
                        int status ) ...
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

status

The status that you want to set.

Library:

None.

Description:

The `_RESMGR_STATUS()` macro sets the *status* member in the `resmgr_context_t` structure.

The resource manager library uses this status when it returns the value from `_RESMGR_NPARTS()` for an I/O or connect function, such as:

[MsgReply](#) (p. 1775) (*ctp->rcvid*, *ctp->status*, *ctp->iiov*, *num*).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

resmgr_unbind()

Disassociate an OCB from an open request

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_unbind( resmgr_context_t * ctp);
```

Arguments:

ctp

A pointer to a [resmgr_context_t](#) (p. 2598) structure that the resource-manager library uses to pass context information between functions.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *resmgr_unbind()* function removes a binding in the internal resources manager database (which maps client connections to server OCB pointers). The binding must previously have been bound by [resmgr_open_bind\(\)](#) (p. 2636)

The binding is reference counted; if multiple connections are established with the same binding, the binding is freed only when the last connection is removed.

You should use [MsgInfo\(\)](#) (p. 1745) to fill the `info` structure in *ctp* with information about which client mapping to retrieve.

Returns:

-1

Failure.

0

Success.

Errors:

ESRCH

The binding can't be located in the resource manager table.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

rewind()

Rewind a file stream to the beginning of the file

Synopsis:

```
#include <stdio.h>

void rewind( FILE *fp );
```

Arguments:

fp

The file stream that you want to rewind.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *rewind()* function rewinds the file stream specified by *fp* to the beginning of the file. It's equivalent to calling [fseek\(\)](#) (p. 772) like this:

```
fseek( fp, 0L, SEEK_SET );
```

except that the error indicator for the stream is cleared.

Examples:

This example shows how you might implement a two-pass assembler:

```
#include <stdio.h>
#include <stdlib.h>

void assemble_pass( FILE *fp, int passno )
{
    printf( "Pass %d\n", passno );

    /* Do more work on the fp */
    switch( passno ) {
    case 1:
        /* do the first-pass work */
        break;

    case 2:
        /* do the second-pass work */
        break;

    default:
        break;
    }
}
```

```
}  
  
int main( void )  
{  
    FILE *fp;  
  
    fp = fopen( "program.s", "r" );  
    if( fp != NULL ) {  
        assemble_pass( fp, 1 );  
        rewind( fp );  
  
        assemble_pass( fp, 2 );  
        fclose( fp );  
  
        return EXIT_SUCCESS;  
    }  
  
    puts( "Error opening program.s" );  
    return EXIT_FAILURE;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

rewinddir()

Rewind a directory stream to the start of the directory

Synopsis:

```
#include <sys/types.h>
#include <dirent.h>

void rewinddir( DIR * dirp );
```

Arguments:

dirp

A pointer to the directory stream of the directory to rewind.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *rewinddir()* function rewinds the directory stream specified by *dirp* to the start of the directory. The directory stream will now refer to the current state of the directory, as if the calling thread had called *opendir()* (p. 1913) again.



The result of using a directory stream after one of the *exec*()* (p. 550) or *spawn*()* (p. 3089) family of functions is undefined. After a call to *fork()* (p. 685), either the parent or the child (but not both) can continue processing the directory stream, using the *readdir()* (p. 2514) and *rewinddir()* functions. If both the parent and child processes use these functions, the result is undefined. Either (or both) processes may use *closedir()* (p. 338).

Examples:

List all the files in a directory, create a new file, and then list the directory contents again:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdlib.h>

int main( void )
```

```

{
    DIR *dirp;
    struct dirent *direntp;
    int filedes;

    dirp = opendir( "/home/fred" );
    if( dirp != NULL ) {
        printf( "Old directory listing\n" );
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;
            printf( "%s\n", direntp->d_name );
        }

        filedes = creat( "/home/fred/file.new",
            S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
        close( filedes );

        rewinddir( dirp );
        printf( "New directory listing\n" );
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;
            printf( "%s\n", direntp->d_name );
        }
        closedir( dirp );
    }

    return EXIT_SUCCESS;
}

```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Rgetsockname()

Get the name of a socket (via a SOCKS server)

Synopsis:

```
#include <sys/socket.h>

int Rgetsockname( int s,
                  struct sockaddr * name,
                  int * namelen );
```

Arguments:

s

The file descriptor of the socket whose name you want to get.

name

A pointer to a `sockaddr` object where the function can store the socket's name.

namelen

A pointer to a `socklen_t` object that initially indicates the amount of space pointed to by *name*. The function updates *namelen* to contain the actual size of the name (in bytes).

Library:

`libsocks`

Use the `-l socks` option to `qcc` to link against this library.

Description:

The *Rgetsockname()* function is a cover function for [getsockname\(\)](#) (p. 960) — the difference is that *Rgetsockname()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Classification:

[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

rindex()

Find the last occurrence of a character in a string

Synopsis:

```
#include <strings.h>

char *rindex( const char *s,
              int c );
```

Arguments:

- s**
The string you want to search. This string must end with a null (\0) character. The null character is considered to be part of the string.
- c**
The character you're looking for.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *rindex()* function returns a pointer to the last occurrence of the character *c* in the string *s*.

Returns:

A pointer to the character, or NULL if the character doesn't occur in the string.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

rint(), rintf(), rintl()

Round to the nearest integral value

Synopsis:

```
#include <math.h>

double rint ( double x );
float rintf ( float x );
long double rintl( long double x );
```

Arguments:

x

The number that you want to round.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *rint()*, *rintf()*, and *rintl()* functions return the integral value nearest *x* in the direction of the current rounding mode.

If the current rounding mode rounds toward negative infinity, then *rint()* is identical to [floor\(\)](#) (p. 671). If the current rounding mode rounds toward positive infinity, then *rint()* is identical to [ceil\(\)](#) (p. 260).

Returns:

An integer (represented as a double precision number) nearest *x* in the direction of the current rounding mode (*IEEE754*).

If <i>x</i> is:	<i>rint()</i> returns:
±Infinity	<i>x</i>
NAN	NAN



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function,

and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;

    a = 0.7 ;
    b = rint(a);
    printf("Round Native mode %f -> %f \n", a, b);

    return(0);
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Rlisten()

Listen for connections on a socket (via a SOCKS server)

Synopsis:

```
#include <sys/socket.h>

int Rlisten( int s,
            int backlog );
```

Arguments:

s

The descriptor for the socket that you want to listen on. You can create a socket by calling [socket\(\)](#) (p. 3075).

backlog

The maximum length that the queue of pending connections may grow to.

Library:

libsocks

Use the `-l socks` option to `qcc` to link against this library.

Description:

The *Rlisten()* function is a cover function for [listen\(\)](#) (p. 1523) — the difference is that *Rlisten()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Classification:

[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

rmkdir()

Delete an empty directory

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

int rmkdir( const char* path );
```

Arguments:

path

The path of the directory that you want to delete. This path can be relative to the current working directory, or an absolute path.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *rmkdir()* function removes (deletes) the specified directory. The directory must not contain any files or directories.



If the directory is the current working directory of any process, *rmkdir()* returns `-1` and sets *errno* (p. 539) to `EINVAL`. If the directory is the root directory, the effect of this function depends on the filesystem.

The space occupied by the directory is freed, making it inaccessible, if its link count becomes zero and no process has the directory open (*opendir()* (p. 1913)). If a process has the directory open when the last link is removed, the `.` and `..` entries are removed and no new entries can be created in the directory. In this case, the directory will be removed when all references to it have been closed (*closedir()* (p. 338)).

When successful, *rmkdir()* marks *st_ctime* and *st_mtime* for update in the parent directory.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EACCES

Search permission is denied for a component of *path*, or write permission is denied on the parent directory of the directory to be removed.

EBUSY

The directory named by *path* can't be removed because it's being used by another process, and the implementation considers this to be an error.

EEXIST

The *path* argument names a directory that isn't empty.

ELOOP

Too many levels of symbolic links.

ENAMETOOLONG

The argument *path* exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.

ENOENT

The specified *path* doesn't exist, or *path* is an empty string.

ENOSYS

The *rmdir()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of *path* isn't a directory.

ENOTEMPTY

The *path* argument names a directory that isn't empty.

EROFS

The directory entry to be removed resides on a read-only filesystem.

Examples:

To remove the directory called /home/terry:

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void )
{
    (void)rmdir( "/home/terry" );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

round(), roundf(), roundl()

Round a number to the closest integer

Synopsis:

```
#include <math.h>

double round( double x );
float roundf( float x );
long double roundl( long double x );
```

Arguments:

x

The number you want to round.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *round()*, *roundf()*, and *roundl()* functions return *x* rounded to the nearest integer *n*, rounding halfway cases away from zero, regardless of the current rounding direction (i.e. returning the value with larger magnitude if $|n - x| == 1/2$).

An application wishing to check for error situations should set [errno](#) (p. 539) to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if *errno* is non-zero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an error has occurred.

Returns:

The rounded value.

Examples:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    double num1, result1;
```

```
for (num1 = 3.0; num1 <= 4.0; num1 += 0.1)
{
    result1 = round (num1);
    printf ("Rounding %f gives %f\n", num1, result1);
}

return (EXIT_SUCCESS);
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

ROUTE

System packet forwarding database

Synopsis:

```
#include <sys/socket.h>
#include <net/if.h>
#include <net/route.h>

int socket( PF_ROUTE,
            SOCK_RAW,
            int family );
```

Description:

QNX TCP/IP provides some packet routing facilities.

The socket manager maintains a routing information database that's used in selecting the appropriate network interface when transmitting packets.

A user process (or possibly multiple cooperating processes) maintains this database by sending messages over a special kind of socket. This supplants fixed-size *ioctl()*s used in earlier releases. Routing table changes may be carried out only by the superuser.

This interface may spontaneously emit routing messages in response to external events, such as receipt of a redirect or of a failure to locate a suitable route for a request. The message types are described in greater detail below.

Routing database entries

Routing database entries come in two flavors: for a specific host or for all hosts on a generic subnetwork (as specified by a bit mask and value under the mask). The effect of wildcard or default routing may be achieved by using a mask of all zeros. There may be hierarchical routes.

When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it's ready for traffic. Normally the protocol specifies the route through each interface as a “direct” connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (i.e. the packet is forwarded).

Routing packets

When routing a packet, the kernel attempts to find the most specific route matching the destination. (If there are two different mask and value-under-the-mask pairs that

match, the more specific is the one with more bits in the mask. A route to a host is regarded as being supplied with a mask of as many ones as there are bits in the destination). If no entry is found, the destination is declared to be unreachable, and a routing-miss message is generated if there are any listeners on the routing control socket described below.

A wildcard routing entry is specified with a zero destination address value and a mask of all zeroes. Wildcard routes are used when the system fails to find other routes matching the destination. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

Routing control messages

To open the channel for passing routing control messages, use the socket call shown in the synopsis above.

The family parameter may be `AF_UNSPEC`, which provides routing information for all address families, or it can be restricted to a specific address family. There can be more than one routing socket open per system.

Messages are formed by a header followed by a small number of `sockaddrs` (with variable length) interpreted by position and delimited by the new length entry in the `sockaddr`. An example of a message with four addresses might be a redirect: Destination, Netmask, Gateway, and Author of the redirect. The interpretation of which addresses are present is given by a bit mask within the header; the sequence is least-significant to most-significant bit within the vector.

Any messages sent to the socket manager are returned, and copies are sent to all interested listeners. The interface provides the process ID for the sender. To distinguish between outstanding messages, the sender may use an additional sequence field. However, message replies may be lost when socket manager buffers are exhausted.

The interface may reject certain messages, as indicated in the `rtm_errno` field.

This error occurs:	If:
EEXIST	Requested to duplicate an existing entry.
ESRCH	Requested to delete a nonexistent entry.
ENOBUFS	Insufficient resources were available to install a new route.

In the current implementation, all routing processes run locally, and the values for `rtm_errno` are available through the normal [errno](#) (p. 539) mechanism, even if the routing reply message is lost.

A process may avoid the expense of reading replies to its own messages by calling [setsockopt\(\)](#) (p. 2909), to turn off the `SO_USELOOPBACK` option at the `SOL_SOCKET`

level. A process may ignore all messages from the routing socket by doing a [shutdown\(\)](#) (p. 2942) system call for further input.

If a route is in use when it's deleted, the routing entry is marked down and removed from the routing table, but the resources associated with it won't be reclaimed until all references to it are released. User processes can obtain information about the routing entry to a specific destination by using a RTM_GET message or by calling [sysctl\(\)](#) (p. 3311).

The messages are:

```
#define RTM_ADD      0x1  /* Add Route */
#define RTM_DELETE  0x2  /* Delete Route */
#define RTM_CHANGE  0x3  /* Change Metrics, Flags, or Gateway */
#define RTM_GET      0x4  /* Report Information */
#define RTM_LOSING   0x5  /* Kernel Suspects Partitioning */
#define RTM_REDIRECT 0x6  /* Told to use different route */
#define RTM_MISS     0x7  /* Lookup failed on this address */
#define RTM_RESOLVE  0xb  /* request to resolve dst to LL addr */
#define RTM_NEWADDR  0xc  /* address being added to iface */
#define RTM_DELADDR  0xd  /* address being removed from iface */
#define RTM_IFINFO   0xe  /* iface going up/down etc. */
```

A message header consists of one of the following:

```
struct rt_msghdr {
    u_short rtm_msglen; /* skip over non-understood msgs */
    u_char  rtm_version; /* future binary compatibility */
    u_char  rtm_type; /* message type */
    u_short rtm_index; /* index for associated ifp */
    int     rtm_flags; /* flags, incl kern & message, e.g. DONE */
    int     rtm_addrs; /* bitmask identifying sockaddrs in msg */
    pid_t   rtm_pid; /* identify sender */
    int     rtm_seq; /* for sender to identify action */
    int     rtm_errno; /* why failed */
    int     rtm_use; /* from rtentry */
    u_long  rtm_inits; /* which metrics we're initializing */
    struct  rt_metrics rtm_rmx; /* metrics themselves */
};

struct if_msghdr {
    u_short ifm_msglen; /* to skip over non-understood msgs */
    u_char  ifm_version; /* future binary compatibility */
    u_char  ifm_type; /* message type */
    int     ifm_addrs; /* like rtm_addrs */
    int     ifm_flags; /* value of ifm_flags */
    u_short ifm_index; /* index for associated ifp */
    struct  if_data ifm_dat /* statistics and other data about if */
};

struct ifa_msghdr {
    u_short ifam_msglen; /* to skip over non-understood msgs */
    u_char  ifam_version; /* future binary compatibility */
    u_char  ifam_type; /* message type */
    int     ifam_addrs; /* like rtm_addrs */
    int     ifam_flags; /* value of ifa_flags */
    u_short ifam_index; /* index for associated ifp */
    int     ifam_metric; /* value of ifa_metric */
};
```

The RTM_IFINFO message uses an if_msghdr header. The RTM_NEWADDR and RTM_DELADDR messages use an ifa_msghdr header. All other messages use the rt_msghdr header.

The metrics structure is:

```
struct rt_metrics {
    u_long rmx_locks; /* Kernel must leave these values alone */
    u_long rmx_mtu; /* MTU for this path */
    u_long rmx_hopcount; /* max hops expected */
    u_long rmx_expire; /* lifetime for route, e.g. redirect */
    u_long rmx_recvpipe; /* inbound delay-bandwidth product */
    u_long rmx_sendpipe; /* outbound delay-bandwidth product */
    u_long rmx_ssthresh; /* outbound gateway buffer limit */
    u_long rmx_rtt; /* estimated round trip time */
    u_long rmx_rttvar; /* estimated rtt variance */
    u_long rmx_pksent; /* packets sent using this route */
};
```

Flags include the values:

```
#define RTF_UP      0x1  /* route usable */
#define RTF_GATEWAY 0x2  /* destination is a gateway */
#define RTF_HOST    0x4  /* host entry (net otherwise) */
```

```

#define RTF_REJECT      0x8    /* host or net unreachable */
#define RTF_DYNAMIC    0x10   /* created dynamically (by redirect) */
#define RTF_MODIFIED    0x20   /* modified dynamically (by redirect) */
#define RTF_DONE        0x40   /* message confirmed */
#define RTF_MASK        0x80   /* subnet mask present */
#define RTF_CLONING     0x100  /* generate new routes on use */
#define RTF_XRESOLVE    0x200  /* external daemon resolves name */
#define RTF_LLINFO      0x400  /* generated by ARP or ESIS */
#define RTF_STATIC      0x800  /* manually added */
#define RTF_BLACKHOLE    0x1000 /* just discard pkts (during updates) */
#define RTF_PROTO2      0x4000 /* protocol specific routing flag */
#define RTF_PROTO1      0x8000 /* protocol specific routing flag */

```

Specifiers for metric values in *rmx_locks* and *rtm_inits* are:

```

#define RTV_MTU          0x1    /* init or lock _mtu */
#define RTV_HOPCOUNT    0x2    /* init or lock _hopcount */
#define RTV_EXPIRE       0x4    /* init or lock _expire */
#define RTV_RPIPE        0x8    /* init or lock _recvpipes */
#define RTV_SPIPE        0x10   /* init or lock _sendpipes */
#define RTV_SSTHRESH     0x20   /* init or lock _sssthresh */
#define RTV_RTT          0x40   /* init or lock _rtt */
#define RTV_RTTVAR       0x80   /* init or lock _rttvar */

```

Specifiers for which addresses are present in the messages are:

```

#define RTA_DST          0x1    /* destination sockaddr present */
#define RTA_GATEWAY      0x2    /* gateway sockaddr present */
#define RTA_NETMASK      0x4    /* netmask sockaddr present */
#define RTA_GENMASK      0x8    /* cloning mask sockaddr present */
#define RTA_IFP          0x10   /* interface name sockaddr present */
#define RTA_IFA          0x20   /* interface addr sockaddr present */
#define RTA_AUTHOR       0x40   /* sockaddr for author of redirect */
#define RTA_BRD          0x80   /* for NEWADDR, */
                                /* broadcast or p-p dest addr */

```

Examples:

Use the following code to set the default route:

```

#include <sys/socket.h>
#include <sys/uio.h>
#include <unistd.h>
#include <net/route.h>
#include <netinet/in.h>
#include <stdio.h>
#include <libgen.h>
#include <arpa/inet.h>
#include <process.h>
#include <errno.h>

struct my_rt
{
    struct rt_msghdr rt;
    struct sockaddr_in dst;
    struct sockaddr_in gate;
    struct sockaddr_in mask;
};

int main(int argc, char **argv)
{
    int s;
    struct rt_msghdr *rtm;
    struct sockaddr_in *dst, *gate, *mask;
    struct my_rt my_rt;

    if(argc < 2)
    {
        fprintf(stderr,
            "Usage: %s: <ip_addr_of_default_gateway>\n",
            basename(argv[0]));
        return 1;
    }

    if((s = socket(AF_ROUTE, SOCK_RAW, 0)) == -1)
    {
        perror("socket");
    }

```

```

        return 1;
    }

    memset(&my_rt, 0x00, sizeof(my_rt));

    rtm = &my_rt.rtm;

    dst = &my_rt.dst;
    gate = &my_rt.gate;
    mask = &my_rt.mask;

    rtm->rtm_type = RTM_ADD;
    rtm->rtm_flags = RTF_UP | RTF_GATEWAY | RTF_STATIC;
    rtm->rtm_msglen = sizeof(my_rt);
    rtm->rtm_version = RTM_VERSION;
    rtm->rtm_seq = 1234;
    rtm->rtm_addr = RTA_DST | RTA_GATEWAY | RTA_NETMASK;
    rtm->rtm_pid = getpid();

    dst->sin_len = sizeof(*dst);
    dst->sin_family = AF_INET;

    mask->sin_len = sizeof(*mask);
    mask->sin_family = AF_INET;

    gate->sin_len = sizeof(*gate);
    gate->sin_family = AF_INET;
    inet_aton(argv[1], &gate->sin_addr);

AGAIN:
    if(write(s, rtm, rtm->rtm_msglen) < 0)
    {
        if(errno == EEXIST && rtm->rtm_type == RTM_ADD)
        {
            rtm->rtm_type = RTM_CHANGE;
            goto AGAIN;
        }

        perror("write");
        return 1;
    }
    return 0;
}

```

Rrcmd()

Execute a command on a remote host (via a SOCKS server)

Synopsis:

```
int Rrcmd( char ** ahost,  
           int inport,  
           const char * locuser,  
           const char * remuser,  
           const char * cmd,  
           int * fd2p );
```

Arguments:

ahost

The name of the host that you want to execute the command on. If the function can find the host, it sets **ahost* to the standard name of the host.

inport

The well-known Internet port on the host, where the server resides.

locuser

The user ID on the local machine.

remuser

The user ID on the remote machine.

cmd

The command that you want to execute.

fd2p

See [rcmd\(\)](#) (p. 2491).

Library:

libsocks

Use the -l socks option to qcc to link against this library.

Description:

The *Rrcmd()* function is a cover function for [rcmd\(\)](#) (p. 2491) — the difference is that *Rrcmd()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

A valid socket descriptor; or -1 if an error occurs and a message is printed to standard error.

Classification:

[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

rresvport()*, *rresvport_af()

Obtain a socket with a privileged address

Synopsis:

```
#include <unistd.h>

int rresvport( int * port );
int rresvport_af( int * port,
                  int af );
```

Arguments:

port

An address in the privileged port space. Privileged Internet ports are those in the range 0 to 1023. Only the superuser may bind this type of address to a socket.

af

(*rresvport_af()* only) The address family; see <sys/socket.h>.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *rresvport()* function returns a descriptor to a socket with an address in the privileged port space. The *rresvport_af()* function is similar, but you can use it for address families other than AF_INET.

The [ruserok\(\)](#) (p. 2696) function is used by servers to authenticate clients requesting service with [rcmd\(\)](#) (p. 2491). All three functions are present in the same file and are used by the rshd server (see the *Utilities Reference*), among others.

The *rresvport()* and *rresvport_af()* functions obtain a socket with a privileged address bound to it. This socket can be used by *rcmd()* and several other functions.

Returns:

A valid, bound socket descriptor, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

The error code EAGAIN is overloaded to mean “All network ports in use.”

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Rselect()

Check for descriptors that are ready for reading or writing (via a SOCKS server)

Synopsis:

```
int Rselect( int width,
             fd_set * readfds,
             fd_set * writefds,
             fd_set * exceptionfds,
             struct timeval * timeout );
```

Arguments:

width

The number of descriptors to check in the given sets. Only the descriptors from 0 through (*width*-1) in the descriptor sets are examined. Therefore, the value of *width* must be at least as large as:

(highest valued file descriptor in the sets) +1

readfds

NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for reading. The function replaces the set with the file descriptors that are actually ready for reading.

writefds

NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for writing. The function replaces the set with the file descriptors that are actually ready for writing.

exceptionfds

NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that have an exceptional condition pending. The function replaces the set with the file descriptors that actually have an exceptional condition pending.

timeout

NULL, or a pointer to a `timeval` that specifies how long to wait for the selection to complete.

Library:

libsocks

Use the -l socks option to qcc to link against this library.

Description:

The *Rselect()* function is a cover function for [select\(\)](#) (p. 2791) — the difference is that *Rselect()* does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

The number of ready descriptors in the descriptor sets, 0 if the *timeout* expired, or -1 if an error occurs ([errno](#) (p. 539) is set).

Classification:

[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

rsrdbmgr_attach()

Reserve a system resource for a process

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_attach( rsrc_request_t * list,
                    int count );
```

Arguments:

list

An array of [rsrc_request_t](#) (p. 2675) structures that describe the resources that you want to reserve; see below.

count

The number of entries in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The resource database manager allocates and keeps track of system resources i.e. it manages these resources. The system resources currently tracked are:

- memory
- IRQs
- DMA channels
- I/O ports
- PCI memory

You can also track and access arbitrarily defined resources by using a string name as an identifier.

Major and minor device numbers are handled with separate [rsrdbmgr_devno_attach\(\)](#) (p. 2688) and [rsrdbmgr_devno_detach\(\)](#) (p. 2691) functions.

There are two main functions that drivers can use to communicate with the resource database:

- `rsrddbmgr_attach()`
- `rsrddbmgr_detach()` (p. 2686)

The `rsrddbmgr_attach()` function reserves a resource range(s) from the database of available resources for a process. Other processes can't reserve this resource range until the resource is returned to the system (usually with the `rsrddbmgr_detach()` call). The requested resources are returned in a *list* of `rsrc_request_t` structures with the *start* and *end* fields filled in. The number of resources requested is specified in *count*.



- In order to successfully use the `rsrddbmgr_attach` and `rsrddbmgr_detach()` functions, your process must have the `PROCmgr_AID_RSRCDBMGR` ability enabled. For more information, see `procmgr_ability()` (p. 2141).
- Reserving the resources doesn't give you access to them; you still have to use `mmap()` (p. 1656), `InterruptAttach()` (p. 1147), or another means.

When you're finished with the resource, you must return it to the system. The easiest way to return the resource is to call `rsrddbmgr_detach()` (p. 2686) with the same *start*, *end*, and type (via the *flags* field) that were issued for the resource.

rsrc_request_t structure

The resource requests structure looks like this:

```
typedef struct _rsrc_request {
    uint64_t    length;
    uint64_t    align;
    uint64_t    start;
    uint64_t    end;
    uint32_t    flags;
    uint32_t    zero[2];
    const char  *name;
} rsrc_request_t;
```

The members include:

length

The length of the resource that you want to reserve. You must set this member.

align

The alignment of the resource.

start, end

The range of resource that you want to reserve.

flags

The type of the resource, as well as flags that affect the request. You must either set this member to be one of the following resource types (defined in `<sys/rsrddbmgr.h>`), or set it to `RSRCDBMGR_FLAG_NAME` and fill in the *name* field:

- `RSRCDBMGR_DMA_CHANNEL` — DMA channel
- `RSRCDBMGR_IO_PORT` — I/O port address
- `RSRCDBMGR_IRQ` — Interrupt address
- `RSRCDBMGR_MEMORY` — Memory address
- `RSRCDBMGR_PCI_MEMORY` — PCI memory address

You can OR in the following flags (also defined in `<sys/rsrddbmgr.h>`):

- `RSRCDBMGR_FLAG_ALIGN` — the contents of the *align* field are valid, and the requested resource starts with the given alignment.
- `RSRCDBMGR_FLAG_LIST` — this is one item in a list to search.
- `RSRCDBMGR_FLAG_NAME` — the *name* field is valid, and is a system name.
- `RSRCDBMGR_FLAG_RANGE` — the contents of the *start* and *end* fields are valid, and the requested resource is in the range *start* to *end*, inclusive.
- `RSRCDBMGR_FLAG_SHARE` — other processes can have access to an allocated resource.
- `RSRCDBMGR_FLAG_TOPDOWN` — start the search for a free resource block from *end*. If you also set `RSRCDBMGR_FLAG_RANGE`, this flag makes the search start from the *end* of the available range.

name

A pointer to the name of the resource class (NULL by default). This field is used only if you set `RSRCDBMGR_FLAG_NAME` in the *flags* field. The name must not start with a leading slash (/) and should be in the form:

type[/subtype[/subsubtype]].

The numeric constants map to the string-based resource names as follows:

Flag	Name
<code>RSRCDBMGR_MEMORY</code>	memory
<code>RSRCDBMGR_IRQ</code>	irq
<code>RSRCDBMGR_IO_PORT</code>	io

Flag	Name
RSRDCBMGR_DMA_CHANNEL	dma
RSRDCBMGR_PCI_MEMORY	pcimemory

For example, normal I/O Ports (RSRDCBMGR_IO_PORT) are mapped to the string `io`. If you had a special class of `io` ports, you could access it with the name `io/myspecialio`. The subtype components are specific instances of the hierarchy, and as such, requests that are made higher up (e.g. a request for `io`) attempt to give out resources that aren't part of these specific instances.

Returns:

EOK, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The resource request can't be filled.

EINVAL

Invalid argument.

ENOMEM

Insufficient memory to allocate internal data structures.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:



When you start the system, the startup code and special programs that know how to probe the hardware call [rsrddbmgr_create\(\)](#) (p. 2681) to register the hardware in the resource database. The following examples don't do this seeding, so they'll fail with an error code of `EINVAL`.

Example 1

```
/*
 * Request one DMA Channel, with length 1, from the
 * entire range of available DMA channel resources.
 */
#include <stdio.h>
#include <sys/rsrddbmgr.h>
```

```

#include <sys/rsrdbmsg.h>

int main(int argc, char **argv) {
    int count;
    rsrc_request_t req;

    memset(&req, 0, sizeof(req));
    req.length = 1;
    req.flags = RSRDBMGR_DMA_CHANNEL;
    count = 1;

    if (rsrdbmgr_attach( &req, count) == -1) {
        perror("Problem attaching to resource ");
        exit(1);
    }

    printf("You can use DMA channel 0x%llx \n",
           req.start);

    ...
    /* Do something with the acquired resource */
    ...

    /* To return the resource to the database: */
    if (rsrdbmgr_detach( &req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }

    return(0);
}

```

Example 2

```

/*
 * Request memory that's 4-byte aligned
 * and has a length of 50.
 */

#include <stdio.h>
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int main(int argc, char **argv) {
    int count;
    rsrc_request_t req;

    memset(&req, 0, sizeof(req));
    req.align = 4;
    req.length = 50;
    req.flags = RSRDBMGR_FLAG_ALIGN | RSRDBMGR_MEMORY;
    count = 1;

    if (rsrdbmgr_attach(&req, count) == -1) {
        perror("Problem attaching to resource ");
        exit(1);
    }

    printf("You can use memory from 0x%llx 0x%llx inclusive. \n",
           req.start, req.end );

    ...
    /* Do something with the acquired resource */
    ...

    /* To return the resource to the database: */
    if (rsrdbmgr_detach( &req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }

    return(0);
}

```

Example 3

```

/*

```

```

* Request two resources:
* I/O port 0 and an IRQ in the range 10-12
* from the available resources.
*/
#include <stdio.h>
#include <sys/rsrddbmgr.h>
#include <sys/rsrddbmsg.h>

int main(int argc, char **argv) {
    int count;
    rsrc_request_t req[2];

    memset(req, 0, 2*sizeof(*req));
    req[0].start = 0;
    req[0].end = 0;
    req[0].length = 1;
    req[0].flags = RSRDCBMGR_FLAG_RANGE | RSRDCBMGR_IO_PORT;

    req[1].start = 10;
    req[1].end = 12;
    req[1].length = 1;
    req[1].flags = RSRDCBMGR_FLAG_RANGE | RSRDCBMGR_IRQ;
    count = 2;

    if (rsrddbmgr_attach(req, count) == -1) {
        perror("Problem attaching to resource ");
        exit(1);
    }

    printf("You can use io-port 0x%llx \n",
           req[0].start);
    printf("You can use irq 0x%llx \n",
           req[1].start);

    ...
    /* Do something with the acquired resource */
    ...

    /* To return the resource to the database: */
    if (rsrddbmgr_detach(req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }

    return(0);
}

```

Example 4

This example uses names instead of the constants for the resource type:

```

rsrc_alloc_t  alloc;
rsrc_request_t ralloc;

//Create a new resource

memset(&alloc, 0, sizeof(alloc));
alloc.start = 0x20000000;
alloc.end = 0x27ffffff;
alloc.flags = RSRDCBMGR_FLAG_NAME;
alloc.name = "io/Port_S0";
printf("Allocing %p %p %p \n", &alloc, &alloc.name, alloc.name);

if (rsrddbmgr_create (&alloc, 1) == -1) {
    perror("create() 1 failed \n");
    return 1;
}

// Attach to the new resource, any range with 32 length

memset( &ralloc, 0, sizeof(ralloc));
ralloc.start = ralloc.end = 0;
ralloc.length = 32;
ralloc.flags = RSRDCBMGR_FLAG_NAME;
ralloc.name = "io/Port_S0";

if (rsrddbmgr_attach (&ralloc, 1) == -1) {
    perror("attach() 2 failed \n");
}

```

```
    return 1;
}
// Do something with the resource
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

rsrdbmgr_create()

Create a system resource

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrcdbmgr_create( rsrc_alloc_t *item,
                    int count );
```

Arguments:

item

An array of [rsrc_alloc_t](#) (p. 2681) structures that describe the resources that you want to create; see below.

count

The number of entries in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *rsrdbmgr_create()* function creates one or more system resources. If the function completes successfully, *count* resources are returned in *item*.

rsrc_alloc_t structure

The structure of a basic resource request looks like this:

```
typedef struct _rsrc_alloc {
    uint64_t start;
    uint64_t end;
    uint32_t flags;
    const char *name;
} rsrc_alloc_t;
```

The members include:

start, end

The resource range.

flags

The type of the resource, as well as flags that affect the request. You must set this member to be one of the following resource types (defined in `<sys/rsrddbmgr.h>`):

- `RSRCDBMGR_DMA_CHANNEL` — DMA channel
- `RSRCDBMGR_IO_PORT` — I/O port address
- `RSRCDBMGR_IRQ` or `RSRCMGR_IRQ` — interrupt address
- `RSRCDBMGR_MEMORY` — Memory address
- `RSRCDBMGR_PCI_MEMORY` — PCI memory address

You can OR in the following bits (also defined in `<sys/rsrddbmgr.h>`):

- `RSRCDBMGR_FLAG_NAME` — the *name* field is valid, and is a system name.
- `RSRCDBMGR_FLAG_NOREMOVE` — don't remove this resource when the process dies.
- `RSRCDBMGR_FLAG_RSVP` — create and reserve a resource with a higher priority than other resources. The resource is given out only when there are no other valid ranges available.

name

NULL, or a pointer to the name of the resource class. This field is used only if you set `RSRCDBMGR_FLAG_NAME` in the *flags* field. For more information, see [rsrddbmgr_attach\(\)](#) (p. 2674).

You must set all the members.

Returns:

EOK, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EAGAIN**

The resource request can't be created.

EINVAL

Invalid argument.

ENOMEM

Insufficient memory to allocate internal data structures.

Examples:

```

/*
 * Create two resources:
 * 0-4K memory allocation and 5 DMA channels.
 */
#include <stdio.h>
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int main(int argc, char **argv) {
    rsrc_alloc_t alloc[2];

    memset(alloc, 0, 2* sizeof(*alloc));
    alloc[0].start = 0;
    alloc[0].end = 4*1024;
    alloc[0].flags = RSRDBMGR_MEMORY;

    alloc[1].start = 1;
    alloc[1].end = 5;
    alloc[1].flags = RSRDBMGR_DMA_CHANNEL;

    /* Allocate resources to the system. */
    if (rsrdbmgr_create( alloc, 2 ) == -1) {
        perror("Problem creating resources \n");
        exit(1);
    }

    ...
    /* Do something with the created resource */
    ...

    /* Remove the allocated resources. */
    rsrcdbmgr_destroy ( alloc, 2 );

    return(0);
}

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

rsrdbmgr_destroy()

Destroy a system resource

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_destroy( rsrc_alloc_t *item,
                     int count );
```

Arguments:

item

An array of [rsrc_alloc_t](#) (p. 2681) structures that describe the resources that you want to destroy. For more information about this structure, see the documentation for *rsrdbmgr_create()*.

count

The number of entries in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *rsrdbmgr_destroy()* function removes *count* system resources that are defined in the array *item*.

Returns:

EOK

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EINVAL

Invalid argument, or the resource is in use.

ENOMEM

Insufficient memory to allocate internal data structures.

Examples:

See the example in [rsrddbmgr_create\(\)](#) (p. 2681).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

rsrdbmgr_detach()

Return a system resource to the resource database

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_detach( rsrc_request_t *list,
                    int count );
```

Arguments:

list

An array of [rsrc_request_t](#) (p. 2675) structures that describe the resources that you want to return. For information about this structure, see the documentation for [rsrdbmgr_attach\(\)](#).

count

The number of entries in the array.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [rsrdbmgr_detach\(\)](#) function returns *count* resources in *list* to the database of available system resources. You must return the resource with the same *start*, *end*, and *flags* (type) that were issued for the resource when it was allocated with [rsrdbmgr_attach\(\)](#) (p. 2674).



In order to successfully use the [rsrdbmgr_attach](#) and [rsrdbmgr_detach\(\)](#) functions, your process must have the `PROCmgr_AID_RSRCDBMGR` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

EOK

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EINVAL

Invalid argument, or the resource is in use by a process, isn't found in the database, or can't be returned to the system.

ENOMEM

Insufficient memory to allocate internal data structures.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

See the examples in [rsrddbmgr_attach\(\)](#) (p. 2674).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

rsrdbmgr_devno_attach()

Get major and minor device numbers

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

dev_t rsrdbmgr_devno_attach( const char * name,
                             int minor_request,
                             int flags );
```

Arguments:

name

The name of the class of devices that you want to get the major number for. This string can be anything, but various names are defined in in `<sys/ftype.h>`; see “[Class names](#) (p. 2689),” below.

minor_request

The minor device number that you want to reserve, or -1 to let the system assign the next available minor number.

flags

Presently, there are no flags; pass zero for this argument.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *rsrdbmgr_devno_attach()* reserves a device number that consists of:

- a major number that corresponds to the given device class. If there isn't already a major number associated with the class, a new major number is assigned to it.
- a minor number that's based on *minor_request*. If *minor_request* is -1, the function returns the first free minor number in the specified class.

There's a maximum of 64 major numbers (0 through 63) on the system, and a maximum of 1024 minor numbers (0 through 1023) per major number.

Major and minor numbers are used only by resource managers and are exposed through the *rdev* member of the `iofunc_attr_t` (p. 1221) structure, and correspondingly the *st_rdev* member of the `stat` structure. They aren't required for proper operation; on simple devices, an entry will be simulated for you.

Class names

As mentioned above, the name of the class of devices can be anything. The following class names are defined in `<sys/ftype.h>`:

Constant	Value	Class
<code>_MAJOR_PATHMGR</code>	"pathmgr"	Used only by the path manager
<code>_MAJOR_DEV</code>	"dev"	Devices in <code>/dev</code> with only one instance (e.g. <code>/dev/tty</code>)
<code>_MAJOR_BLK_PREFIX</code>	"blk-"	All block devices (e.g. <code>/dev/hd[0-9]*</code> would be "blk-hd")
<code>_MAJOR_CHAR_PREFIX</code>	"char-"	All character devices (e.g. <code>/dev/ser[0-9]*</code> would be "char-ser")
<code>_MAJOR_FSYS</code>	"fsys"	All filesystems

Returns:

A `dev_t` object that contains the major and minor numbers, or -1 if an error occurs (*errno* (p. 539) is set).

You can extract the major and minor number values from the `dev_t` object by using the *major()* and *minor()* macros defined in `<sys/types.h>`. For more information, see the documentation for *stat()* (p. 3164).

Errors:

EINVAL

Invalid argument.

Examples:

```
#include <sys/rsrddbmgr.h>
#include <sys/rsrddbmsg.h>

char    *dev_name;
int     myminor_request, flags=0;
dev_t   major_minor;
```

```
major_minor = rsrcdbmgr_devno_attach
               ( dev_name, myminor_request, flags );

...

rsrcdbmgr_devno_detach( major_minor, flags );
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

If your application calls this function, it must run as `root`.

rsrdbmgr_devno_detach()

Detach a major and minor number

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_devno_detach( dev_t devno,
                           int flags );
```

Arguments:

devno

A `dev_t` object that was returned by [rsrdbmgr_devno_attach\(\)](#) (p. 2688).

flags

Presently, there are no flags; pass zero for this argument.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *rsrdbmgr_devno_detach()* detaches device number that was attached with [rsrdbmgr_devno_attach\(\)](#) (p. 2688).

Returns:

EOK

Success.

-1

An error occurred.

Examples:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

dev_t dev_num;
```

```
int    flags=0;

...

rsrddbmgr_devno_detach( dev_num, flags );
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

rsrddbmgr_query_name()

Query the resource database

Synopsis:

```

#include <sys/rsrddbmgr.h>
#include <sys/rsrddbmsg.h>

int rsrddbmgr_query_name( rsrc_alloc_t *list,
                        int listcnt,
                        int start,
                        pid_t pid );
                        char *name );
                        unsigned type );

```

Arguments:

list

NULL, or an array of [rsrc_alloc_t](#) (p. 2681) structures that the function can fill with information about the resources that it finds. For more information about this structure, see the documentation for [rsrddbmgr_create\(\)](#).

listcnt

The number of entries in the array.

start

The index that you want to start searching at.

pid

Which blocks to query:

- -1: query all blocks.
- 0: query all free blocks.
- > 0: query blocks owned by the process with the given *pid*.

name

The name of the resource to query. For more information, see [rsrddbmgr_attach\(\)](#) (p. 2674).

type

The subtype to query; 0, or one of the following:

- RSRCDMGR_DMA_CHANNEL — DMA channel
- RSRCDMGR_IO_PORT — I/O port address
- RSRCDMGR_IRQ — Interrupt address
- RSRCDMGR_MEMORY — Memory address
- RSRCDMGR_PCI_MEMORY — PCI memory address

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `rsrcdbmgr_query_name()` function queries the database for *listcnt* count of *type* resources in use, beginning at the index *start*. If you make the query with a non-NULL *list*, then the function stores a maximum of found *listcnt* resources in the array.

Returns:

If *list* is NULL or *listcnt* is 0, then the function returns the number of resources of *type* in the database.

If *list* is non-NULL, then the function returns the number of *type* resources available in the system.

If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

Errors:**EINVAL**

Invalid argument

ENOMEM

Insufficient memory to allocate internal data structures.

Examples:

List all of the memory resource blocks available in the system:

```
rsrc_alloc_t  list[20];
int           size, count = 0, start = 0;

while (1) {
    count = rsrcdbmgr_query_name( list, 20, start, -1, "memory", 0);
    if (count == -1)
        break;

    size = min( count-start, 20 ); /* In case more than 20 blocks
                                   were returned. */
}
```

```
printf( "Retrieved %d of a possible %d resource blocks", \
        size, count);

for (count=0; count<size; count++) {
    printf( "RSRC[%d] Start %d End %d \n", \
            start+count, list[count].start, list[count].end);
}
start += size; /* Loop again, in case there are more than
                20 blocks. */
}
```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ruserok()

Check the identity of a remote host

Synopsis:

```
#include <unistd.h>

int ruserok( char * rhost,
             int superuser,
             char * ruser,
             char * luser );
```

Arguments:

rhost

The name of the remote host, as returned by [gethostbyaddr\(\)](#) (p. 866).

superuser

Nonzero if the local user is the superuser, zero otherwise.

ruser

The name of the remote user.

luser

The name of the local user.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The *ruserok()* routine checks the identity of a remote host. It's used by servers to authenticate clients requesting service with [rcmd\(\)](#) (p. 2491).

The *rcmd()*, [resvport\(\)](#) (p. 2670), and *ruserok()* functions are used by the `rshd` server (see the *Utilities Reference*), among others.

The *ruserok()* function takes a remote host's name (as returned by the *gethostbyaddr()* routine), two user names, and a flag indicating whether the local user's name is that of the superuser. Then, if the user *isn't* the superuser, it checks the file `/etc/hosts.equiv` (described in the *Utilities Reference*).

If that lookup isn't done, or is unsuccessful, the `.rhosts` file in the local user's home directory is checked to see if the request for service is allowed. If this file is owned by anyone other than the user or the superuser, or if it's writable by anyone other than the owner, the check automatically fails.

If the local domain obtained from [gethostname\(\)](#) (p. 883) is the same as the remote domain, only the machine name need be specified.

Returns:

0

The machine name is listed in the `hosts.equiv` file, or the host and remote username were found in the `.rhosts` file.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

The error code `EAGAIN` is overloaded to mean “All network ports in use.”

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Chapter 21

S

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
S_ISBLK()	stat() (p. 3164)
S_ISCHR()	stat() (p. 3164)
S_ISDIR()	stat() (p. 3164)
S_ISFIFO()	stat() (p. 3164)
S_ISLNK()	stat() (p. 3164)
S_ISNAM()	stat() (p. 3164)
S_ISREG()	stat() (p. 3164)
S_ISSOCK()	stat() (p. 3164)
S_TYPEISMQ()	stat() (p. 3164)
S_TYPEISSEM()	stat() (p. 3164)
S_TYPEISSHM()	stat() (p. 3164)
shutdown_classify()	shutdown_system() (p. 2944)
shutdown_display()	shutdown_system() (p. 2944)
shutdown_done()	shutdown_system() (p. 2944)
shutdown_error()	shutdown_system() (p. 2944)
shutdown_process()	shutdown_system() (p. 2944)
shutdown_progress()	shutdown_system() (p. 2944)
shutdown_prompt()	shutdown_system() (p. 2944)
SIGEV_CLEAR_CRITICAL()	sigevent (p. 2961)
SIGEV_GET_TYPE()	sigevent (p. 2961)
SIGEV_INTR_INIT()	sigevent (p. 2961)

For information about:	See:
<i>SIGEV_MAKE_CRITICAL()</i>	sigevent (p. 2961)
<i>SIGEV_NONE_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_PULSE_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_SIGNAL_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_SIGNAL_VALUE_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_SIGNAL_CODE_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_SIGNAL_THREAD_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_THREAD_INIT()</i>	sigevent (p. 2961)
<i>SIGEV_UNBLOCK_INIT()</i>	sigevent (p. 2961)
<i>_SLOG_SETCODE()</i>	slogf() (p. 3063)
<i>SOCKCREDSIZE()</i>	UNIX (p. 3577)
<i>SUN_LEN()</i>	UNIX (p. 3577)

The following have been deprecated:

Instead of:	Use:
<i>scalbf()</i>	scalbn() (p. 2707)
SCTP	No replacement
<i>sctp_bindx()</i>	No replacement
<i>sctp_connectx()</i>	No replacement
<i>sctp_freeladdrs()</i>	No replacement
<i>sctp_freepaddrs()</i>	No replacement
<i>sctp_getladdrs()</i>	No replacement
<i>sctp_getpaddrs()</i>	No replacement
<i>sctp_peeloff()</i>	No replacement
<i>sctp_recvmmsg()</i>	No replacement
<i>sctp_sendmsg()</i>	No replacement
<i>setpriority()</i>	pthread_setschedparam() (p. 2400) or SchedSet() (p. 2780)
<i>significand()</i>	scalbn (p. 2707)(<i>x</i> , -ilogb (p. 1057)(<i>x</i>))

Instead of:	Use:
<i>significandf()</i>	scalbnf (p. 2707)(<i>x</i> , -ilogbf (p. 1057)(<i>x</i>))
<i>spin_destroy()</i>	pthread_spin_destroy() (p. 2422)
<i>spin_init()</i>	pthread_spin_init() (p. 2424)
<i>spin_lock()</i>	pthread_spin_lock() (p. 2426)
<i>spin_trylock()</i>	pthread_spin_trylock() (p. 2428)
<i>spin_unlock()</i>	pthread_spin_unlock() (p. 2430)
<i>SyncCreate()</i>	SyncTypeCreate() (p. 3305)
<i>SyncCreate_r()</i>	SyncTypeCreate_r() (p. 3305)

Note the following:

- [setprio\(\)](#) (p. 2890) is for compatibility with QNX 4 programs; use [pthread_setschedparam\(\)](#) (p. 2400) or [SchedSet\(\)](#) (p. 2780)

This chapter describes the API components whose names begin with “S”.

sbrk()

Set the allocation break value

Synopsis:

```
#include <unistd.h>

void* sbrk( int increment );
```

Arguments:

increment

The amount by which to increase the current break value. This increment may be positive or negative.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *break* value is the address of the first byte of unallocated memory. When a program starts execution, the break value is placed following the code and constant data for the program. As memory is allocated, this pointer advances when there is no free block large enough to satisfy an allocation request. The *sbrk()* function sets a new break value for the program by adding the value of *increment* to the current break value. Newly allocated space is set to zero.

The variable `_amblksiz` (p. 151) (defined in `<stdlib.h>`) contains the default increment. This value may be changed by a program at any time.

Returns:

The previous break value, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EAGAIN

The total amount of system memory available for allocation to this process is temporarily insufficient. This may occur although the space requested is less than the maximum data segment size.

ENOMEM

The requested change allocated more space than allowed, or it caused a memory allocation conflict.

Examples:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    void* prev_brk;

    prev_brk = sbrk( 0x3100 );
    printf( "Original break value: %p\n", prev_brk );

    prev_brk = sbrk( 0x0200 );
    printf( "After sbrk( 0x3100 ): %p\n", prev_brk );

    prev_brk = sbrk( 0x0 );
    printf( "After sbrk( 0x0200 ); %p\n", prev_brk );

    return EXIT_SUCCESS;
}
```

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Don't use *brk()* and *sbrk()* with any other memory functions (such as [malloc\(\)](#) (p. 1569), [mmap\(\)](#) (p. 1656), and [free\(\)](#) (p. 717)). The *brk()* function assumes that the heap is contiguous; in QNX Neutrino, memory is returned to the system by the heap, causing the heap to become sparse. The QNX Neutrino *malloc()* function is based on *mmap()*, and not on *brk()*.

The *sbrk()* function has been used in specialized cases where no other memory allocation function provided the same capability. Use *mmap()* instead because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

The value of the argument to *sbrk()* is rounded up for alignment with eight-byte boundaries.

scalb()

Load the exponent of a radix-independent floating point number

Synopsis:

```
#include <math.h>

double scalb( double x,
              double n );
```

Arguments:

x

The floating point number that you want to multiply by the exponent.

n

The exponent to apply to the radix of the machine's floating-point arithmetic.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute $x \times r^n$, where r is the radix of the machine's floating point arithmetic and n is a finite number. When r is 2, `scalb()` is equivalent to `ldexp()` (p. 1506).



We recommend that you use `scalbn()` (p. 2707) because it computes by manipulating exponents, instead of using mock multiplications or additions.

Returns:

$x \times r^n$



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` (p. 539) to 0, call the function, and then check `errno` again. These functions don't change `errno` if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;

    a = 10;
    b = 2;
    c = scalb(a, b);
    d = sqrt(c/a);
    printf("Radix of machines fp arithmetic is %f \n", d);
    printf("So %f = %f * (%f ^ %f) \n", c, a, d, b);

    return(0);
}
```

produces the output:

```
Radix of machines fp arithmetic is 2.000000
So 40.000000 = 10.000000 * (2.000000 ^ 2.000000)
```

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

scalbn(), scalbnf()

Load the exponent of a radix-independent floating point number

Synopsis:

```
#include <math.h>

double scalbn ( double x,
                int  n );

float  scalbnf ( float x,
                int  n );
```

Arguments:

x

The floating point number that you want to multiply by the exponent.

n

The exponent to apply to the radix of the machine's floating-point arithmetic.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The *scalbn()* and *scalbnf()* functions compute $x \times r^n$, where *r* is the radix of the machine's floating point arithmetic.

Returns:

$x \times r^n$

When a correct value would cause an underflow, these functions return 0.0.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;

    a = 10;
    b = 2;
    c = scalbn(a, b);
    d = sqrt(c/a);
    printf("Radix of machines fp arithmetic is %f \n", d);
    printf("So %f = %f * (%f ^ %f) \n", c, a, d, b);

    return(0);
}
```

produces the output:

```
Radix of machines fp arithmetic is 2.000000
So 40.000000 = 10.000000 * (2.000000 ^ 2.000000)
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_scalloc()

Allocate space for an array

Synopsis:

```
#include <malloc.h>

void* _scalloc( size_t size );
```

Arguments:

size

The number of bytes to allocate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `_scalloc()` functions allocate space for an array of length *size* bytes. Each element is initialized to 0.

You must use `\_sfree\(\)` (p. 2923) to deallocate the memory allocated by `_scalloc()`.

Returns:

A pointer to the start of the allocated memory, or `NULL` if there's insufficient memory available or if the *size* is zero.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

scandir()

Scan a directory

Synopsis:

```
#include <sys/types.h>
#include <sys/dir.h>

int scandir( const char * dirname,
             struct direct * (* namelist[]),
             int (*select)(const struct dirent *),
             int (*compar)(const struct dirent **,const struct
             dirent **) );
```

Arguments:

dirname

The name of the directory that you want to scan.

namelist

A pointer to a location where *scandir()* can store a pointer to the array of directory entries (of type `struct direct *`) that it builds.

select

A pointer to a user-supplied subroutine that *scandir()* calls to select which entries to included in the array. The *select* routine is passed a pointer to a directory entry (a `struct dirent` (p. 422)) and should return a nonzero value if the directory entry is to be included in the array.

If *select* is `NULL`, all the directory entries are included.

compar

A pointer to a user-supplied subroutine that's passed to *qsort()* (p. 2476) to sort the completed array. If this pointer is `NULL`, the array isn't sorted.

You can use *alphasort()* (p. 149) as the *compar* parameter to sort the array alphabetically.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `scandir()` function reads the directory *dirname* and builds an array of pointers to directory entries, using `malloc()` (p. 1569) to allocate the space. The `scandir()` function returns the number of entries in the array, and stores a pointer to the array in the location referenced by *namelist*.

Don't confuse the `direct` and `dirent` structures:



- The entries in the array are pointers to `direct` structures.
- The argument to the `select` function is a pointer to a `dirent` structure.
- The arguments to the `compar()` function are used as pointers to `direct` structures.

The `struct direct` is defined as:

```
struct direct {
    unsigned long d_fileno;
    unsigned short d_reclen;
    unsigned short d_namlen;
    char d_name[1];
};
```

You can deallocate the memory allocated for the array by calling `free()` (p. 717). Free each pointer in the array, and then free the array itself.

Returns:

The number of entries in the array, or -1 if the directory can't be opened for reading, or `malloc()` can't allocate enough memory to hold all the data structures.

Classification:

Legacy Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

scanf()

Scan formatted input from stdin

Synopsis:

```
#include <stdio.h>

int scanf( const char* format,
           ... );
```

Arguments:

format

A string that controls the format of the input, as described below. The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `scanf()` function scans input from *stdin* under control of the *format* argument, assigning values to the remaining arguments.



The `scanf()` function is generally considered unsafe for string handling; it's safer to use [fgets\(\)](#) (p. 652) to get a line of input and then use [sscanf\(\)](#) (p. 3162) to process the input.

Format control string

The format control string consists of zero or more *format directives* that specify what you consider to be acceptable input data. Subsequent arguments are pointers to various types of objects that the function assigns values to as it processes the format string.

A format directive can be a sequence of one or more whitespace characters or:

multibyte characters

Any character in the *format* string, other than a whitespace character or the percent character (%), that isn't part of a conversion specifier.

conversion specifiers

A sequence of characters in the *format* string that begins with a percent character (%) and is followed by:

- an optional *assignment suppression indicator*: the asterisk character (*)
- an optional decimal integer that specifies the *maximum field width* to be scanned for the conversion
- an optional *type length* specification; one of h, L, or l
- a character that specifies the type of conversion to be performed; one of the characters: c, d, e, f, g, i, n, o, p, s, u, X, x, [

As each format directive in the format string is processed, the directive may successfully complete, fail because of a lack of input data, or fail because of a matching error as defined by the directive.

If end-of-file is encountered on the input data before any characters that match the current directive have been processed (other than leading whitespace, where permitted), the directive fails for lack of data.

If end-of-file occurs after a matching character has been processed, the directive is completed (unless a matching error occurs), and the function returns without processing the next directive.

If a directive fails because of an input character mismatch, the character is left unread in the input stream.

Trailing whitespace characters, including newline characters, aren't read unless matched by a directive. When a format directive fails, or the end of the format string is encountered, the scanning is completed, and the function returns.

When one or more whitespace characters (space, horizontal tab \t, vertical tab \v, form feed \f, carriage return \r, newline or linefeed \n) occur in the format string, input data up to the first non-whitespace character is read, or until no more data remains. If no whitespace characters are found in the input data, the scanning is complete, and the function returns.

An ordinary character in the format string is expected to match the same character in the input stream.

Conversion specifiers

A conversion specifier in the format string is processed as follows:

- For conversion types other than [, c and n, leading whitespace characters are skipped.
- For conversion types other than n, all input characters, up to any specified maximum field length, that can be matched by the conversion type are read and converted to the appropriate type of value; the character immediately following the last

character to be matched is left unread; if no characters are matched, the format directive fails.

- Unless you specify the assignment suppression indicator (*), the result of the conversion is assigned to the object pointed to by the next unused argument (if assignment suppression was specified, no argument is skipped); the arguments must correspond in number, type and order to the conversion specifiers in the format string.

Type length specifiers

A type length specifier affects the conversion as follows:

- **hh** causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** format conversion to assign the converted value to an object of type **signed char** or **unsigned char**.
- **h** causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** (integer) format conversion to assign the converted value to an object of type **short** or **unsigned short**.
- **j** causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** conversion to assign the converted value to an object of type **intmax_t** or **uintmax_t**.
- **l** (“el”) causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** (integer) conversion to assign the converted value to an object of type **long** or **unsigned long**.
- **l** (“el”) causes an **a**, **A**, **e**, **E**, **f**, **F**, **g** or **G** conversion to assign the converted value to an object of type **double**.
- **l** (“el”) causes a **c**, **s** or **[** conversion to assign the converted value to an object of type **wchar_t**.
- **ll** (double “el”) causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** format conversion to assign the converted value to an object of type **long long** or **unsigned long long**.
- **L** causes an **a**, **A**, **e**, **E**, **f**, **F**, **g** or **G** conversion to assign the converted value to an object of type **long double**.
- **t** causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** conversion to assign the converted value to an object of type **ptrdiff_t** or to the corresponding **unsigned** type.
- **z** causes a **d**, **i**, **o**, **u**, **x**, **X** or **n** conversion to assign the converted value to an object of type **size_t** or to the corresponding signed integer type.

Conversion type specifiers

The valid conversion type specifiers are:

a, A, e, E, f, F, g or G

A floating-point number, infinity, or NaN, all of which have a format as expected by [strtod\(\)](#) (p. 3253). The argument is assumed to point to an object of type **float**.

c

Any sequence of characters in the input stream of the length specified by the field width, or a single character if you don't specify a field width. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence, *without* a terminating NUL character ('\\0'). For a single character assignment, a pointer to a single object of type `char` is sufficient.

When an `l` ("el") qualifier is present, a sequence of characters are converted from the initial shift state to `wchar_t` wide characters as if by a call to [mbrtowc\(\)](#) (p. 1587). The conversion state is described by a `mbstate_t` object.

d

A decimal integer with a format as expected by [strtol\(\)](#) (p. 3262) and a *base* of 10. The argument is assumed to point to an object of type `int`.

i

An optionally signed integer with a format as expected by [strtol\(\)](#) (p. 3262) and a *base* of 0. The argument is assumed to point to an object of type `int`.

n

No input data is processed. Instead, the number of characters that have already been read is assigned to the object of type `int` that's pointed to by the argument. The number of items that have been scanned and assigned (the return value) isn't affected by the `n` conversion type specifier.

o

An optionally signed octal integer with a format as expected by [strtoul\(\)](#) (p. 3264) and a *base* of 8. The argument is assumed to point to an object of type `int`.

p

A hexadecimal integer, as described for `x` conversions below. The converted value is taken as a `void *` and then assigned to the object pointed to by the argument.

If the sequence contains more than 8 digits, the conversion is performed only on the last 8 digits in the sequence.

s

A sequence of non-whitespace characters. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence of `char`, `signed char` or `unsigned char` and a terminating NUL character, which the conversion operation adds.

When an `l` (“`l`”) qualifier is present, a sequence of characters are converted from the initial shift state to `wchar_t` wide characters as if by a call to [mbrtowc\(\)](#) (p. 1587). The conversion state is described by a `mbstate_t` object.

u

An unsigned decimal integer, consisting of one or more decimal digits. The argument is assumed to point to an object of type `unsigned int`.

x, X

A hexadecimal integer, with a format as expected by [strtoul\(\)](#) (p. 3264) when *base* is 16. The argument is assumed to point to an object of type `unsigned`.

[

Matches the *scanset*, a nonempty sequence of characters. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating NUL character, which the conversion operation adds.

When an `l` (“`l`”) qualifier is present, a sequence of characters are converted from the initial shift state to `wchar_t` wide characters as if by a call to [mbrtowc\(\)](#) (p. 1587) with *mbstate* set to 0. The argument is assumed to point to the first element of a `wchar_t` array of sufficient size to contain the sequence and a terminating NUL character, which the conversion operation adds.

The conversion specification includes all characters in the scanlist between the beginning `[` and the terminating `]`. If the conversion specification starts with `[^`, the scanlist matches all the characters that *aren't* in the scanlist. If the conversion specification starts with `[]` or `[^]`, the `]` is included in the scanlist. (To scan for `]` only, specify `%[]`.)

A `-` in the scanlist that isn't the first character, nor the second where the first character is a `^`, nor the last character, defines a range of characters to be matched. This range consists of characters numerically greater than or equal to the character before the `-`, and numerically less than or equal to the character after the `-`.

%

A `%` character (The entire specification is `%%`).

A conversion type specifier of `%` is treated as a single ordinary character that matches a single `%` character in the input data. A conversion type specifier other than those listed above causes scanning to terminate, and the function to return with an error.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning stopped by reaching the end of the input stream before storing any values.

Examples:

The line:

```
scanf( "%s%f%3hx%d", name, &hexnum, &decnum )
```

with input:

```
some_string 34.555e-3 abc1234
```

copies "some_string" into the array *name*, skips 34.555e-3, assigns 0xabc to *hexnum* and 1234 to *decnum*. The return value is 3.

The program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char string1[80], string2[80];

    memset( string1, 0, 80 );
    memset( string2, 0, 80 );

    scanf( "[%abcdefghijklmnopqrstuvwyz"
           "ABCDEFGHJKLMNOPQRSTUVWXYZ ]%*2s[^\n]",
           string1, string2 );

    printf( "%s\n", string1 );
    printf( "%s\n", string2 );

    return EXIT_SUCCESS;
}
```

with input:

```
They may look alike, but they don't perform alike.
```

assigns "They may look alike" to *string1*, skips the comma (the "%*2s" matches only the comma; the following blank terminates that field), and assigns " but they don't perform alike." to *string2*.

To scan a date in the form "Friday March 26 1999":

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    int day, year;
    char weekday[10], month[12];
    int retval;

    memset( weekday, 0, 10 );
    memset( month, 0, 12 );

    retval = scanf( "%10s %12s %d %d",
```

```
        weekday, month, &day, &year );
if( retval != 4 ) {
    printf( "Error reading date.\n" );
    printf( "Format is: Friday March 26 1999\n" );

    return EXIT_FAILURE;
}

printf( "weekday: %s\n", weekday );
printf( "month: %s\n", month );
printf( "day: %d\n", day );
printf( "year: %d\n", year );

return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

sched_getparam()

Get the current scheduling parameters of a process

Synopsis:

```
#include <sched.h>

int sched_getparam( pid_t pid,
                    struct sched_param *param );
```

Arguments:

pid

The ID of the process whose scheduling parameters you want to get, or 0 to get them for the current process.

param

A pointer to a [sched_param](#) (p. 2731) that the function fills with the scheduling parameters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sched_getparam()* function gets the current scheduling parameters (including the original and current priorities) of the process specified by *pid*, and puts them in the *sched_param* structure pointed to by *param*.

If *pid* is zero, function gets the parameters for the calling process.

Returns:

0

Success

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EPERM**

The calling process doesn't have sufficient privilege to get the scheduling parameters.

ESRCH

The process *pid* doesn't exist.

Examples:

```
#include <sched.h>
#include <stdio.h>

#define PRIORITY_ADJUSTMENT 5

int main (void)
{
    int max_priority;
    struct sched_param param;

    /* Find out the MAX priority for the FIFO Scheduler */
    max_priority = sched_get_priority_max(SCHED_FIFO);

    /* Find out what the current priority is. */
    sched_getparam(0, &param);

    printf("The assigned priority is %d.\n", param.sched_priority);
    printf("The current priority is %d.\n", param.sched_curpriority);

    param.sched_priority = ((param.sched_curpriority +
        PRIORITY_ADJUSTMENT) <= max_priority) ?
        (param.sched_curpriority + PRIORITY_ADJUSTMENT) : -1;

    if (param.sched_priority == -1)
    {
        printf(
            "Cannot increase the priority by %d. Try a smaller value\n",
            PRIORITY_ADJUSTMENT);
        return(0);
    }

    sched_setscheduler (0, SCHED_FIFO, &param);

    sched_getparam(0, &qparam);
    printf("The newly assigned priority is %d.\n",
        param.sched_priority);
    printf("The current priority is %d.\n",
        param.sched_curpriority);
    return(0);
}
```

Classification:

POSIX 1003.1 PS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

In a multithreaded program, *sched_getparam()* returns the scheduling parameters for thread 1 in the process *pid*, or for the *calling thread* if *pid* is 0.

sched_get_priority_adjust()

Calculate the allowable priority for the scheduling policy

Synopsis:

```
#include <sched.h>

int sched_get_priority_adjust( int prio,
                              int policy,
                              int adjust );
```

Arguments:

prio

The original priority value. If negative, the priority of the calling thread is used.

policy

The scheduling policy being used. The valid arguments are listed in [sched_get_priority_max\(\)](#) (p. 2725). If *policy* is SCHED_NOCHANGE, the function uses the algorithm of the calling thread.

adjust

The priority change, relative to *prio*. A value of +10 results in a final priority of *prio*+10, provided that this amount of adjustment is allowed.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `sched_get_priority_adjust()` function calculates the requested priority change relative to another thread and returns the allowable value.

This function makes it easier for you to set relative priorities in order to ensure proper precedence.

Returns:**>0**

The allowed priority value. The value will never exceed the range of values allowed by [sched_get_priority_min\(\)](#) (p. 2727) and [sched_get_priority_max\(\)](#) (p. 2725).

<0

Failure; the negative of the [errno](#) (p. 539) value.

Errors:**EINVAL**

The value of the *policy* parameter doesn't represent a defined scheduling policy.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sched_get_priority_max()

Get the maximum priority for the scheduling policy

Synopsis:

```
#include <sched.h>

int sched_get_priority_max( int policy );
```

Arguments:

policy

The scheduling policy, which must be one of:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
- **SCHED_RR** — similar to **SCHED_FIFO**, except that threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- **SCHED_OTHER** — currently the same as **SCHED_RR**.
- **SCHED_SPORADIC** — sporadic scheduling. For more information, see [pthread_attr_setschedpolicy\(\)](#) (p. 2223), and “Scheduling policies” in the chapter on the microkernel in the *System Architecture* guide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sched_get_priority_max()* function returns the maximum value for the scheduling policy specified by *policy*.

Returns:

The appropriate minimum for success, or -1 ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of the *policy* parameter doesn't represent a defined scheduling policy.

ENOSYS

The *sched_get_priority_max()* function isn't currently supported.

Examples:

See [sched_getparam\(\)](#) (p. 2720).

Classification:

[POSIX 1003.1 PS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sched_get_priority_min()

Get the minimum priority for the scheduling policy

Synopsis:

```
#include <sched.h>

int sched_get_priority_min( int policy );
```

Arguments:

policy

The scheduling policy, which must be one of:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
- **SCHED_RR** — similar to **SCHED_FIFO**, except that threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- **SCHED_OTHER** — currently the same as **SCHED_RR**.
- **SCHED_SPORADIC** — sporadic scheduling. For more information, see [pthread_attr_setschedpolicy\(\)](#) (p. 2223), and “Scheduling policies” in the chapter on the microkernel in the *System Architecture* guide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sched_get_priority_min()* function returns the minimum value for the scheduling policy specified by *policy*.

Returns:

The appropriate minimum for success, or -1 ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of the *policy* parameter doesn't represent a defined scheduling policy.

ENOSYS

The *sched_get_priority_min()* function isn't currently supported.

Classification:

[*POSIX 1003.1 PS*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sched_getscheduler()

Get the current scheduling policy for a process

Synopsis:

```

#include <sched.h>

int sched_getscheduler( pid_t pid );

```

Arguments:

pid

The ID of the process whose scheduling policy you want to find, or zero if you want to get the policy for the current process.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The sched_getscheduler() function gets the current scheduling policy of process *pid*. If *pid* is zero, the scheduling policy of the calling process is returned.

Returns:

The scheduling policy, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

ESRCH

The process *pid* doesn't exist.

Classification:

POSIX 1003.1 PS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

In a multithreaded program, *sched_getscheduler()* returns the scheduling policy for thread 1 in the process *pid*, or for the *calling thread* if *pid* is 0.

sched_param

Structure that describes scheduling parameters

Synopsis:

```
#include <sched.h>

struct sched_param {
    int32_t  sched_priority;
    int32_t  sched_curpriority;
    union {
        int32_t  reserved[8];
        struct {
            int32_t  __ss_low_priority;
            int32_t  __ss_max_repl;
            struct timespec  __ss_repl_period;
            struct timespec  __ss_init_budget;
        }
        __ss;
    }
    __ss_un;
}

#define sched_ss_low_priority    __ss_un.__ss.__ss_low_priority
#define sched_ss_max_repl      __ss_un.__ss.__ss_max_repl
#define sched_ss_repl_period    __ss_un.__ss.__ss_repl_period
#define sched_ss_init_budget    __ss_un.__ss.__ss_init_budget
```

Description:

You'll use the `sched_param` structure when you get or set the scheduling parameters for a thread or process.

You can use these functions to get the scheduling parameters:

- [pthread_attr_getschedparam\(\)](#) (p. 2197)
- [pthread_getschedparam\(\)](#) (p. 2299)
- [sched_getparam\(\)](#) (p. 2720)
- [SchedGet\(\)](#) (p. 2774)

You can use these functions to set the scheduling parameters:

- [pthread_attr_setschedparam\(\)](#) (p. 2221)
- [pthread_setschedparam\(\)](#) (p. 2400)
- [pthread_setschedprio\(\)](#) (p. 2402)
- [sched_setparam\(\)](#) (p. 2738)
- [sched_setscheduler\(\)](#) (p. 2740)
- [SchedSet\(\)](#) (p. 2780)
- [ThreadCreate\(\)](#) (p. 3402)

The members of `sched_param` include:

sched_priority

When you get the scheduling parameters, this member reflects the priority that was assigned to the thread or process. It doesn't reflect any temporary adjustments due to priority inheritance.

When you set the scheduling parameters, set this member to the priority that you want to use. The priority must be between the minimum and maximum values returned by [sched_get_priority_min\(\)](#) (p. 2727) and [sched_get_priority_max\(\)](#) (p. 2725) for the scheduling policy.

As an extension to POSIX in QNX Neutrino 6.6 or later, you can wrap the new priority in one these macros to specify how to handle out-of-range priority requests:

- *SCHED_PRIO_LIMIT_ERROR(priority)* — indicate an error
- *SCHED_PRIO_LIMIT_SATURATE(priority)* — saturate at the maximum allowed priority

If `procnto` was started with a `-P` option ending with `s` or `S`, out-of-range priority requests by default saturate at the maximum allowed value.

sched_curpriority

When you get the scheduling parameters, this member is set to the priority that the thread or process is currently running at. This is the value that the kernel uses when making scheduling decisions.

When you set the scheduling parameters, this member is ignored.

The other members are used with sporadic scheduling. The following `#define` directives create the POSIX names that correspond to those members and should be used instead of accessing members directly.

sched_ss_low_priority

The background or low priority for the thread that's executing.

sched_ss_max_repl

The maximum number of times a replenishment will be scheduled, typically because of a blocking operation. After a thread has blocked this many times, it automatically drops to the low-priority level for the remainder of its execution until its execution budget is replenished.

sched_ss_repl_period

The time that should be used for scheduling the replenishment of an execution budget after being blocked or having overrun the maximum number

of replenishments. This time is used as an offset against the time that a thread is made READY.

sched_ss_init_budget

The time that should be used for the thread's execution budget. As the thread runs at its high-priority level, its execution time is carved out of this budget. Once the budget is entirely depleted, the thread drops to its low-priority level, where, if possible because of priority arrangements, it can continue to run until the execution budget is replenished.



- The *sched_priority* must always be higher than *sched_ss_low_priority*.
- The *sched_ss_max_repl* must be smaller than SS_REPL_MAX.
- The *sched_ss_init_budget* must be smaller than *sched_ss_repl_period*.

For more information, see “Scheduling policies” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

Examples:

This code shows a duty-cycle usage of the sporadic server thread:

```
#include <stdio.h>
#include <errno.h>
#include <sched.h>
#include <pthread.h>
#include <inttypes.h>
#include <sys/syspage.h>
#include <sys/neutrino.h>

/* 50 % duty cycle of 5 secs on 5 secs off */
struct timespec g_init_budget = { 5, 0 };
struct timespec g_repl_period = { 10, 0 };

#define MY_HIGH_PRIORITY 5
#define MY_LOW_PRIORITY 4
#define MY_REPL_PERIOD g_repl_period
#define MY_INIT_BUDGET g_init_budget
#define MY_MAX_REPL 10

#define DUTY_CYCLE_LOOPS 10

/*
 * Run a compute-bound thread (minimal blocking) to show the
 * duty cycle.
 */
void *st_duty_check(void *arg) {
    struct sched_param params;
    uint64_t stime, etime, cps;
    double secs;
    int ret, prio;
    int prevprio, iterations;

    stime = ClockCycles();
    cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    iterations = 0;

    printf("\n");
```

```

    prevprio = -1;
    while(iterations < DUTY_CYCLE_LOOPS) {
        etime = ClockCycles();
        ret = pthread_getschedparam(pthread_self(), &prio,
                                   &params);

        if(ret != 0) {
            printf("pthread_getschedparam() failed %d \n",
                   errno);
            break;
        } else if (prevprio != -1 && prevprio !=
                    params.sched_priority) {
            stime = etime - stime;
            secs = (double)stime / (double)cps;
            printf("pri %d (cur %d) %lld cycles %g secs\n",
                   params.sched_priority,
                   params.sched_curpriority,
                   stime, secs);
            stime = etime;
            iterations++;
        }
        prevprio = params.sched_priority;
    }
    return NULL;
}

int main(int argc, char **argv) {
    struct sched_param params;
    pthread_attr_t attr;
    pthread_t thr;
    int ret;

    /* Set the attribute structure with the sporadic values */
    printf("# Set sporadic attributes ...");
    pthread_attr_init(&attr);
    ret = pthread_attr_setinheritsched(&attr,
                                       PTHREAD_EXPLICIT_SCHED);
    if(ret != 0) {
        printf("pthread_attr_setinheritsched() failed %d \n",
               errno);
        return 1;
    }

    ret = pthread_attr_setschedpolicy(&attr, SCHED_SPORADIC);
    if(ret != 0) {
        printf("pthread_attr_setschedpolicy() failed %d \n",
               ret, errno);
        return 1;
    }

    params.sched_priority = MY_HIGH_PRIORITY;
    params.sched_ss_low_priority = MY_LOW_PRIORITY;
    memcpy(&params.sched_ss_init_budget, &MY_INIT_BUDGET,
          sizeof(MY_INIT_BUDGET));
    memcpy(&params.sched_ss_repl_period, &MY_REPL_PERIOD,
          sizeof(MY_REPL_PERIOD));
    params.sched_ss_max_repl = MY_MAX_REPL;
    ret = pthread_attr_setschedparam(&attr, &params);
    if(ret != 0) {
        printf("pthread_attr_setschedparam() failed %d \n", errno);
        return 1;
    }
    printf("OK\n");

    /* Create a sporadic thread to check the duty cycle */
    printf("# Creating thread duty cycle thread (%d changes) ... ",
           DUTY_CYCLE_LOOPS);
    ret = pthread_create(&thr, &attr, st_duty_check, NULL);
    if(ret != 0) {
        printf("pthread_create() failed %d \n", errno);
        return 1;
    }
    pthread_join(thr, NULL);
    printf("OK\n");

    return 0;
}

```

See also [sched_getparam\(\)](#) (p. 2720).

Classification:

[POSIX 1003.1 PS](#) (p. 46)

sched_rr_get_interval()

Get the execution time limit of a process

Synopsis:

```
#include <sched.h>

int sched_rr_get_interval(
    pid_t pid,
    struct timespec * interval );
```

Arguments:

pid

The process ID whose execution time limit you want to get.

interval

A pointer to a [timespec](#) (p. 3473) structure that the function updates with the process's current execution time limit.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *sched_rr_get_interval()* function updates *interval* with the current execution time limit for the process, *pid*. If *pid* is 0, the current execution time limit for the calling process is returned.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ENOSYS

The *sched_rr_get_interval()* function isn't currently supported.

ESRCH

The process *pid* can't be found.

Classification:

POSIX 1003.1 PS (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sched_setparam()

Change the priority of a process

Synopsis:

```
#include <sched.h>

int sched_setparam(
    pid_t pid,
    const struct sched_param *param );
```

Arguments:

pid

The ID of the process whose priority you want to set, or 0 to set it for the current process.

param

A pointer to a [sched_param](#) (p. 2731) structure whose *sched_priority* member holds the priority that you want to assign to the process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sched_setparam()* function changes the priority of process *pid* to that of the *sched_priority* member in the *sched_param* structure pointed to by *param*. If *pid* is zero, the priority of the calling process is changed.

The *sched_priority* member in *param* must lie between the minimum and maximum values returned by [sched_get_priority_max\(\)](#) (p. 2725) and [sched_get_priority_min\(\)](#) (p. 2727).

By default, the process priority and scheduling policy are inherited from or explicitly set by the parent process. Once running, the child process may change its priority by using this function.



In order to change its priority to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

0

Success

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

The priority isn't a valid priority.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process *pid* doesn't exist.

Classification:

[POSIX 1003.1 PS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

In a multithreaded program, `sched_setparam()` sets the scheduling parameters for thread 1 in the process *pid*, or for the *calling thread* if *pid* is 0.

sched_setscheduler()

Change the priority and scheduling policy of a process

Synopsis:

```
#include <sched.h>

int sched_setscheduler(
    pid_t pid,
    int policy,
    const struct sched_param *param );
```

Arguments:

pid

The ID of the process whose priority and scheduling policy you want to set, or zero if you want to set them for the current process.

policy

The scheduling policy, which must be one of:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
- **SCHED_RR** — similar to **SCHED_FIFO**, except that threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- **SCHED_OTHER** — currently the same as **SCHED_RR**.
- **SCHED_SPORADIC** — sporadic scheduling.

For more information, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

param

A pointer to a [sched_param](#) (p. 2731) structure whose *sched_priority* member holds the priority that you want to assign to the process.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `sched_setscheduler()` function changes the priority of process *pid* to that of the `sched_priority` member in the `sched_param` structure passed as *param*, and the scheduling policy is set to *policy*.

If *pid* is zero, the policy and priority of the calling process are set.

The `sched_priority` member in *param* must lie between the minimum and maximum values returned by [sched_get_priority_max\(\)](#) (p. 2725) and [sched_get_priority_min\(\)](#) (p. 2727).

By default, the process priority and scheduling policy are inherited from or explicitly set by the parent process. Once running, the child process may change its priority by using this function.



In order to change its priority to a value above the maximum permitted for unprivileged processes, your process must have the `PROC_MGR_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The previous scheduling policy, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**EFAULT**

A fault occurred trying to access the buffers provided.

EINVAL

The priority or scheduling policy isn't a valid value.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process *pid* doesn't exist.

Examples:

See [sched_getparam\(\)](#) (p. 2720).

Classification:

[POSIX 1003.1 PS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

In a multithreaded program, *sched_setscheduler()* sets the scheduling policy for thread 1 in the process *pid*, or for the *calling thread* if *pid* is 0.

sched_yield()

Yield to other ready threads at the same priority

Synopsis:

```
#include <sched.h>

int sched_yield( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sched_yield()` function checks to see if other threads, at the same priority as that of the calling thread, are READY to run. If so, the calling thread yields to them and places itself at the end of the READY thread queue. The `sched_yield()` function never yields to a lower-priority thread.

A higher-priority thread always forces a lower-priority thread to yield (that is, preempt) the instant the higher-priority thread becomes ready to run, without the need for the lower-priority thread to give up the processor by calling the `sched_yield()` or [SchedYield\(\)](#) (p. 2783) functions.

The `sched_yield()` function calls the kernel function [SchedYield\(\)](#) (p. 2783), and may be more portable across realtime POSIX systems.



You should avoid designing programs that contain busy wait loops. If you can't avoid them, you can use `sched_yield()` to reduce the system load at a given priority level. Note that a thread that calls `sched_yield()` in a tight loop will spend a great deal of time in the kernel, which will have a small effect on interrupt latency.

Returns:

This function always succeeds and returns zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
```

```
int main( void )
{
    int i;

    for( ;; ) {
        /* Process something... */
        for( i = 0 ; i < 1000 ; ++i )
            fun();

        /* Yield to anyone else at the same priority */
        sched_yield();
    }
    return EXIT_SUCCESS; /* Never reached */
}

int fun()
{
    int i;

    for( i = 0 ; i < 10 ; ++i )
        i += i;

    return( i );
}
```

Classification:

POSIX 1003.1 PSITHR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SchedCtl(), SchedCtl_r()

Control the adaptive partitioning scheduler

Synopsis:

```
#include <sys/sched_aps.h>
#include <sys/neutrino.h>

int SchedCtl( int cmd,
              void *data,
              int length);

int SchedCtl_r( int cmd,
                void *data,
                int length);
```

Arguments:

cmd

The control command that you want to execute; one of:

- [SCHED_APS_QUERY_PARMS](#) (p. 2746)
- [SCHED_APS_SET_PARMS](#) (p. 2750)
- [SCHED_APS_CREATE_PARTITION](#) (p. 2751)
- [SCHED_APS_LOOKUP](#) (p. 2755)
- [SCHED_APS_QUERY_PARTITION](#) (p. 2753)
- [SCHED_APS_JOIN_PARTITION](#) (p. 2756)
- [SCHED_APS_MODIFY_PARTITION](#) (p. 2757)
- [SCHED_APS_PARTITION_STATS](#) (p. 2759)
- [SCHED_APS_OVERALL_STATS](#) (p. 2761)
- [SCHED_APS_MARK_CRITICAL](#) (p. 2762)
- [SCHED_APS_CLEAR_CRITICAL](#) (p. 2763)
- [SCHED_APS_ATTACH_EVENTS](#) (p. 2766)
- [SCHED_APS_QUERY_THREAD](#) (p. 2764)
- [SCHED_APS_ADD_SECURITY](#) (p. 2767)
- [SCHED_APS_QUERY_PROCESS](#) (p. 2771)

data

A pointer to the specific data structure for the command.

length

The size of the structure that *data* points to.

For details about each command and its data, see the sections below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SchedCtl()* and *SchedCtl_r()* kernel calls control the adaptive partitioning scheduler.

This scheduler is optional and is present only if you add `[module=aps]` to your OS image's buildfile. For more information, see the Adaptive Partitioning *User's Guide*. These functions were added in the QNX Neutrino Core OS 6.3.2.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2772) section for details.



You must initialize all of the fields—including reserved ones—in the structures you pass as the *data* argument, by calling (for example) [memset\(\)](#) (p. 1620). You can also use the `APS_INIT_DATA()` macro:

```
APS_INIT_DATA( &data );
```

SCHED_APS_QUERY_PARMS

This command fills in a `sched_aps_info` structure that describes the overall parameters of the adaptive partitioning scheduler:

```
typedef struct {
    uint64_t    cycles_per_ms;
    uint64_t    window_size_cycles;
    uint64_t    window_size2_cycles;
    uint64_t    window_size3_cycles;
    uint32_t    scheduling_policy_flags;
    uint32_t    sec_flags;
    uint32_t    bankruptcy_policy;
    uint16_t    num_partitions;
    uint16_t    max_partitions;
    uint64_t    reserved1;
    uint64_t    reserved2;
} sched_aps_info;
```

The members include:

cycles_per_ms

The number of machine cycles in a millisecond. Use this value to convert the output of the [SCHED_APS_QUERY_PARTITION](#) (p. 2753) command to the time units of your choice.

The value of *cycles_per_ms*:



- might not equal the value of the *cycles_per_sec* member of the system page divided by 1000
 - isn't necessarily in the same units as values returned by *ClockCycles()* on all platforms
-

window_size_cycles

The length, in CPU cycles, of the averaging window used for scheduling. By default, this corresponds to 100 ms.



If you change the tick size of the system at runtime, do so before defining the adaptive partitioning scheduler's window size. That's because QNX Neutrino converts the window size from milliseconds to clock ticks for internal use.

window_size2_cycles

The length, in CPU cycles, of window 2, for reporting only. Typically 10 times the window size.

window_size3_cycles

The length, in CPU cycles, of window 3, for reporting only. Typically 100 times the window size.

scheduling_policy_flags

The set of SCHED_APS_SCHEDPOL_* flags that describe the scheduling policy. For more information, see “[Scheduling policies](#) (p. 2748),” below.

sec_flags

The set of SCHED_APS_SEC_* flags that describe the security options. For more information, see “[Security](#) (p. 2768),” below.

bankruptcy_policy

What to do if a partition exhausts its critical budget; a combination of SCHED_APS_BNKR_* flags (see “[Handling bankruptcy](#) (p. 2748),” below).

num_partitions

The number of partitions defined.

max_partitions

The largest number of partitions that may be created at any time.

Scheduling policies

These flags set options for the adaptive partitioning scheduling algorithm. To set, pass a pointer to an ORed set of these flags with the [SCHED_APS_SET_PARMS](#) (p. 2750) call to *SchedCtl()*:

SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO

Free time is when at least one partition isn't running. Its time becomes free to other partitions that may then run over their budgets.

By default, the scheduler hands out free time to the partition with the highest-priority running thread. That guarantees realtime scheduling behavior (i.e., scheduling strictly by priority) to partitions any time they aren't being limited by some other partition's right to its guaranteed minimum budget. But it also means that one partition is allowed to grab all the free time.

If you set **SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO**, the running partitions share the free time in proportion to the ratios of their budgets. So, one partition can no longer grab all the free time. However, when this flag is set, partitions will see strict priority-scheduling between partitions only when they're consuming less than their CPU budgets.

SCHED_APS_SCHEDPOL_DEFAULT

The default policy, which currently means that **SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO** isn't set. QNX Neutrino sets this at startup.

Scheduling within a partition is always strictly by priority, no matter which of these flags are set.

For more information about adaptive partitioning and BMP, see the Adaptive Partitioning Scheduling Details chapter of the Adaptive Partitioning *User's Guide*.

Handling bankruptcy

Bankruptcy is when critical CPU time billed to a partition exceeds its critical budget. Bankruptcy is always considered to be a design error on the part of the application, but you can configure how the system responds to it.

If the system isn't declaring bankruptcy when you expect it, note that bankruptcy can be declared only if critical time is billed to your partition. Critical time is billed on those timeslices when the following conditions are *all* met:

- The running partition has a critical budget greater than zero.

- The top thread in the partition is marked as running critical, or has received the critical state from receiving a SIG_INTR, a `sigevent` marked as critical, or has just received a message from a critical thread.
- The running partition must be out of percentage-CPU budget.
- There be at least one other partition that is competing for CPU time.

Only then if the critical time, billed over the current averaging window, exceeds a partition's critical budget will the system declare the partition bankrupt.

When the system detects that a partition has gone bankrupt, it always:

- causes that partition to be out-of-budget for the remainder of the current scheduling window
- delivers any `sigevent` that you've specified as a notification of bankruptcy with the [SCHED_APS_ATTACH_EVENTS](#) (p. 2766) command. This occurs at most once per calling `SCHED_APS_ATTACH_EVENTS`.

In addition, you can configure the following responses:

SCHED_APS_BNKR_BASIC

Deliver bankruptcy-notification events and make the partition out-of-budget for the rest of the scheduling window (nominally 100 ms). This is the default.

SCHED_APS_BNKR_CANCEL_BUDGET

Set the offending partition's critical budget to zero, which forces the partition to be scheduled by its percentage CPU budget only. This also means that a second bankruptcy can't occur. This persists until a restart occurs, or you call [SCHED_APS_MODIFY_PARTITION](#) (p. 2757) to set a new critical budget.

SCHED_APS_BNKR_LOG

Not currently implemented.

SCHED_APS_BNKR_REBOOT

Cause the system to crash with a brief message identifying the offending partition. This is the most severe response, suggested for use while testing a product, to make sure bankruptcies are never ignored. You probably shouldn't use this option in your finished product.

SCHED_APS_BNKR_RECOMMENDED

The combination `SCHED_APS_BNKR_CANCEL_BUDGET | SCHED_APS_BNKR_LOG`. We recommend this choice.

To set a choice of bankruptcy-handling options, OR the above `SCHED_APS_BNKR_*` flags and pass a pointer to it as the `bankruptcy_policy` field of the `sched_aps_parms` structure when you call [SCHED_APS_SET_PARMS](#) (p. 2750).

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_QUERY_PARMS command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

The calling thread doesn't meet the security options set (see [SCHED_APS_ADD_SECURITY](#) (p. 2767)). Your process must have the PROCmgr_AID_APS_ROOT ability (see [procmgr_ability\(\)](#) (p. 2141)).

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_SET_PARMS

The command sets the parameters for the overall behavior of the adaptive partitioning scheduler. The *data* argument must be a pointer to a sched_aps_parms structure:

```
typedef struct {
    int16_t window_size_ms;
    int16_t reserved1;
    uint32_t *scheduling_policy_flagsp;
    uint32_t *bankruptcy_policy;
    int32_t reserved2;
    int64_t reserved3;
} sched_aps_parms;
```

The members include:

window_size_ms

The time over which the scheduler is to average CPU cycles and balance the partitions to their percentage budgets as specified by [SCHED_APS_CREATE_PARTITION](#) (p. 2751) If you don't want to set the window size, set this member to -1.

scheduling_policy_flagsp

A pointer to an ORed set of SCHED_APS_SCHEDPOL_* flags that specify the scheduling policy. For more information, see “[Scheduling policies](#)” (p.

2748),” above. If you don't want to change the scheduling policy, set this member to NULL.

bankruptcy_policy

A pointer to an ORing of SCHED_APS_BNKR_* flags, as described under “[Handling bankruptcy](#) (p. 2748),” above. If you don't want to change these flags, set this member to NULL.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_SET_PARMS command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

One of the following:

- SCHED_APS_SEC_PARTITIONS_LOCKED is set.
- SCHED_APS_SEC_ROOT0_OVERALL is set, and you aren't running in the System partition with the PROCMGR_AID_APS_ROOT ability (see [procmgr_ability\(\)](#) (p. 2141)).

For more information, see “[Security](#) (p. 2768),” below.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_CREATE_PARTITION

This command creates a new partition which is considered to be a child of the partition that's calling *SchedCtl()*. The system automatically creates a partition called System (the value of APS_SYSTEM_PARTITION_NAME) with an ID of 0.

The *data* argument for this command must be a pointer to a sched_aps_create_parms structure:

```
typedef struct {
```

```

/* input parms */
char      *name;
uint16_t   budget_percent;
int16_t    critical_budget_ms;
uint32_t   reserved1;
uint64_t   reserved2;
/* output parms */
int16_t    id;
} sched_aps_create_parms;

```

The input members include:

name

The name of the new partition. If *name* is NULL or points to an empty string, *SchedCtl()* assigns a name, in the form Pa, Pb, Pc, and so on. The name must be no longer than APS_PARTITION_NAME_LENGTH, not including the trailing null character, and can't include any slashes (/).

budget_percent

The percentage CPU budget for the new partition. Budgets given to the new partition are subtracted from the parent partition.



Before creating zero-budget partitions, read the cautions in “Setting budgets for resource managers” in the System Considerations chapter of the Adaptive Partitioning *User's Guide*.

critical_budget_ms

The critical budget, in milliseconds, for the partition, or -1 or 0 if you don't want the partition to have a critical budget. Critical budgets don't affect the parent, but are automatically limited to be no bigger than the window size.

The output members include:

id

The created partition's ID number, in the range 0 to the maximum number of partitions – 1 (see the *max_partitions* member of the data from a call to [SCHED_APS_QUERY_PARMS](#) (p. 2746). The System partition's ID number is APS_SYSTEM_PARTITION_ID.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_CREATE_PARTITION command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

SCHED_APS_SEC_PARTITIONS_LOCKED is set, or any of these security conditions are set and not satisfied:

- SCHED_APS_SEC_ROOT_MAKES_PARTITIONS
- SCHED_APS_SEC_SYS_MAKES_PARTITIONS
- SCHED_APS_SEC_NONZERO_BUDGETS
- SCHED_APS_SEC_ROOT_MAKES_CRITICAL
- SCHED_APS_SEC_SYS_MAKES_CRITICAL

For more information, see “[Security](#) (p. 2768),” below.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EDQUOT

The parent partition doesn't have enough budget.

EEXIST

Another partition is already using the given name.

EINVAL

The size of the parameter block doesn't match the size of the expected structure, the partition name is badly formed, or the budget is out of range.

ENAMETOOLONG

The partition name is longer than *APS_PARTITION_NAME_LENGTH* characters.

ENOSPC

The maximum number of partitions already exist.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_QUERY_PARTITION

This command gets information about a given partition. The *data* argument for this command must be a pointer to a *sched_aps_partition_info* structure:

```
typedef struct {
    /* out parms */
    uint64_t    budget_cycles;
    uint64_t    critical_budget_cycles;
    char        name[APS_PARTITION_NAME_LENGTH+1];
    int16_t     parent_id;
```

```
uint16_t    budget_percent;
int32_t     notify_pid;
int32_t     notify_tid;
uint32_t    pinfo_flags;
int32_t     pid_at_last_bankruptcy;
int32_t     tid_at_last_bankruptcy;
int64_t     reserved1;
int64_t     reserved2;
/* input parm */
int16_t     id;
} sched_aps_partition_info;
```

The input members include:

id

The number of the partition you want to query.

The output members include:

budget_cycles

The budget, in cycles. To convert this value to something useful, convert it with the *cycles_per_ms* value from a [SCHED_APS_QUERY_PARMS](#) (p. 2746) command.

critical_budget_cycles

The critical budget, in cycles.

name

The name of the partition.

parent_id

The number of the partition that's the parent of the partition being queried. The System partition's ID number is `APS_SYSTEM_PARTITION_ID`.

budget_percent

The partition's budget, expressed as a percentage.

notify_pid, notify_tid

The process and thread IDs of the thread to be given overload and bankruptcy notifications, or -1 if not set.

pinfo_flags

A set of flag that give extra information about the partition:

- `SCHED_APS_PINFO_BANKRUPTCY_NOTIFY_ARMED`—see [SCHED_APS_ATTACH_EVENTS](#) (p. 2766)
- `SCHED_APS_PINFO_OVERLOAD_NOTIFY_ARMED`—see [SCHED_APS_ATTACH_EVENTS](#) (p. 2766)

pid_at_last_bankruptcy, tid_at_last_bankruptcy

The process and thread IDs at the time of the last bankruptcy, or -1 if there wasn't a previous bankruptcy.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_QUERY_PARTITION command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_LOOKUP

This command finds the partition ID for a given partition name.

The *data* argument for this command must be a *sched_aps_lookup_parms* structure:

```
typedef struct {
    /* input parms */
    char    *name;
    int16_t reserved1;
    /* output parms */
    int16_t  id;
} sched_aps_lookup_parms;
```

The input members include:

name

The name of the partition.

The output members include:

id

The ID number of the partition, if found.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_LOOKUP command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The name wasn't found.

SCHED_APS_JOIN_PARTITION

This command makes the thread specified by the given process and thread IDs becomes a member of the specified partition. This partition also becomes the thread's new home partition, i.e., where it returns after partition inheritance.

The *data* argument for this command must be a pointer to a *sched_aps_join_parms* structure:

```
typedef struct {
    int16_t    id;
    int16_t    reserved1;
    int32_t    pid;
    int32_t    tid;
    int32_t    reserved2;
} sched_aps_join_parms;
```

The members include:

id

The ID number of the partition that the thread is to join.

pid, tid

The process and thread IDs of the thread that you want to join the specified partition:

- If both *pid* and *tid* are zero, the calling thread joins the specified partition.
- If *tid* is -1, the process with ID *pid* joins the partition. This doesn't change the partitions that the process's threads are in; it just sets the partition that the threads run in when they're handling a pulse.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_JOIN_PARTITION command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

The following security options are set but not satisfied:

- SCHED_APS_SEC_ROOT_JOINS
- SCHED_APS_SEC_SYS_JOINS
- SCHED_APS_SEC_PARENT_JOINS
- SCHED_APS_SEC_JOIN_SELF_ONLY

For more information, see “[Security](#) (p. 2768),” below.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure, or the partition with the given ID doesn't exist.

ENOSYS

The adaptive partitioning scheduler isn't installed.

ESRCH

The *pid* and *tid* are invalid.

SCHED_APS_MODIFY_PARTITION

This command changes the parameters of an existing partition. If the new budget's percent value is different from the current, the difference is either taken from, or returned to, the parent partition's budget. The critical time parameter affects only the chosen partition, not its parent. To change just one of new budget or new critical time, set the other to -1.



- You can't use this command to modify the System partition's budget. To increase the size of the System partition, reduce the budget of one of its child partitions.
- Reducing the size of a partition may cause it not to run for the time of an averaging window, as you may have caused it to become temporarily over-budget. However, reducing the critical time doesn't trigger the declaration of bankruptcy.

The *data* argument for this command must be a pointer to a `sched_aps_modify_parms` structure:

```
typedef struct {
    int16_t      id;
    int16_t      new_budget_percent;
    int16_t      new_critical_budget_ms;
    int16_t      reserved1;
    int64_t      reserved2;
    int64_t      reserved3;
} sched_aps_modify_parms;
```

The members include:

id

The ID number of the partition.

new_budget_percent

The new budget for the partition, expressed as a percentage, or -1 if you don't want to change it.

new_critical_budget_ms

The new critical budget, in milliseconds, for the partition, or -1 if you don't want to change it. If the critical budget is greater than the window size, it's considered to be infinite.

Errors:

`SchedCtl()` and `SchedCtl_r()` indicate the following errors for the `SCHED_APS_MODIFY_PARTITION` command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

`SCHED_APS_SEC_PARTITIONS_LOCKED` is set, or the following security options are set and not satisfied:

- `SCHED_APS_SEC_PARENT_MODIFIES`
- `SCHED_APS_SEC_ROOT_MAKES_PARTITIONS`
- `SCHED_APS_SEC_SYS_MAKES_PARTITIONS`
- `SCHED_APS_SEC_NONZERO_BUDGETS`
- `SCHED_APS_SEC_ROOT_MAKES_CRITICAL`
- `SCHED_APS_SEC_SYS_MAKES_CRITICAL`

For more information, see “[Security](#) (p. 2768),” below.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure, or the partition with the given ID doesn't exist.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_PARTITION_STATS

This command returns instantaneous values of the CPU time-accounting variables for a set of partitions. It can fill in data for more than one partition. If the *length* argument to *SchedCtl()* indicates that you've passed the function an array of *sched_aps_partition_stats* structures, *SchedCtl()* fills each element with statistics for a different partition, starting with the partition specified by the *id* field.



To get an accurate picture for the the whole machine it's important to read data for all partitions in one call, since sequential calls to *SCHED_APS_PARTITION_STATS* may come from separate averaging windows.

To determine the number of partitions, use the [SCHED_APS_OVERALL_STATS](#) (p. 2761) command.

The command overwrites the *id* field with the partition number for which data is being returned. It stores -1 in the *id* field of unused elements.

To convert times in cycles into milliseconds, divide them by the *cycles_per_ms* obtained with an [SCHED_APS_QUERY_PARMS](#) (p. 2746) command.

The *data* argument for this command must be a pointer to a *sched_aps_partition_stats* structure, or an array of these structures:

```
typedef struct {
    /* out parms */
    uint64_t    run_time_cycles;
    uint64_t    critical_time_cycles;
    uint64_t    run_time_cycles_w2;
    uint64_t    critical_time_cycles_w2;
    uint64_t    run_time_cycles_w3;
    uint64_t    critical_time_cycles_w3;
    uint32_t    stats_flags;
    uint32_t    reserved1;
    uint64_t    reserved2;
    uint64_t    reserved3;
    /* in parm */
    int16_t     id;
} sched_aps_partition_stats;
```

The members include:

run_time_cycles

The CPU execution time during the last scheduling window.

critical_time_cycles

The time spent running critical threads during the last scheduling window.

run_time_cycles_w2

The CPU time spent during the last *window_size2_cycles*. Window 2 is typically 10 times the length of the averaging window.

critical_time_cycles_w2

The time spent running critical threads during the last *window_size2_cycles*.

run_time_cycles_w3

The CPU time spent during the last *window_size3_cycles*. Window 3 is typically 100 times the length of the averaging window.

critical_time_cycles_w3

The time spent running critical threads during the last *window_size3_cycles*.

stats_flags

A set of the following flags:

- **SCHED_APS_PSTATS_IS_BANKRUPT_NOW** — the critical time used is greater than the critical budget at the time you used the **SCHED_APS_PARTITION_STATS** command.
- **SCHED_APS_PSTATS_WAS_BANKRUPT** — the partition was declared bankrupt sometime since the last restart.

id

This is both an input and output field. As input, it's the ID number of the first partition you want data for. If you've passed an array of **sched_aps_partition_stats** structures, the command fills in the ID number for each partition that it fills in statistics for.

Errors:

SchedCtl() and **SchedCtl_r()** indicate the following errors for the **SCHED_APS_PARTITION_STATS** command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block isn't a multiple of *size(sched_aps_partition_stats)*.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_OVERALL_STATS

This command returns instantaneous values of overall CPU-usage variables and other dynamic scheduler states. The *data* argument for this command must be a pointer to a *sched_aps_overall_stats* structure:

```
typedef struct {
    uint64_t    idle_cycles;
    uint64_t    idle_cycles_w2;
    uint64_t    idle_cycles_w3;
    int16_t     id_at_last_bankruptcy;
    uint16_t     reserved1;
    int32_t     pid_at_last_bankruptcy;
    int32_t     tid_at_last_bankruptcy;
    uint32_t     reserved2;
    uint32_t     reserved3;
    uint64_t     reserved4;
} sched_aps_overall_stats;
```

The members include:

idle_cycles

The time, in cycles, during the last scheduling window where nothing (other than the idle thread) ran. To convert this to the percent idle time, calculate:

$$(100 \times \text{idle\_cycles}) / \text{window\_size\_cycles}$$

idle_cycles_w2

The time spent running idle during the last *window_size2_cycles*. Window 2 is typically 10 times the length of the averaging window.

idle_cycles_w3

The time spent running idle during last *window_size3_cycles*. Window 3 is typically 100 times the length of the averaging window.

id_at_last_bankruptcy

The ID of last bankrupt partition, or -1 if no bankruptcy has occurred.

pid_at_last_bankruptcy, tid_at_last_bankruptcy

The process and thread IDs at last the bankruptcy, or -1 if no bankruptcy has occurred.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_OVERALL_STATS command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_MARK_CRITICAL

This command sets one thread in your process to run as a critical thread whenever it runs. Use a thread ID of zero to set the calling thread to be critical.



In general, it's more useful to send a critical [sigevent](#) (p. 2961) structure to a thread to make it run as a critical thread.

The *data* argument for this command must be a pointer to a *sched_aps_mark_crit_parms* structure:

```
typedef struct {  
    int32_t    pid;  
    int32_t    tid;  
    int32_t    reserved;  
} sched_aps_mark_crit_parms;
```

The members include:

pid

The process ID, or 0 for the calling process.

tid

The thread ID, or 0 for the calling thread.



You can also set up [sigevent](#) (p. 2961) structures that make their receiving threads run as critical.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the `SCHED_APS_MARK_CRITICAL` command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

ESRCH

The specified thread wasn't found.

SCHED_APS_CLEAR_CRITICAL

This command clears the “always run as critical” state set by the [SCHED_APS_CLEAR_CRITICAL](#) (p. 2763) command. Then the thread will run as critical only when it inherits that state from another thread (on receipt of a message).

The *data* argument for this command must be a pointer to a `sched_aps_clear_crit_parms` structure:

```
typedef struct {
    int32_t    pid;
    int32_t    tid;
    int32_t    reserved;
} sched_aps_clear_crit_parms;
```

The members include:

pid

The process ID, or 0 for the calling process.

tid

The thread ID, or 0 for the calling thread.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_CLEAR_CRITICAL command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

ESRCH

The specified thread wasn't found.

SCHED_APS_QUERY_THREAD

This command determines the partition for the given thread and indicates whether or not the thread in your process is marked to run as critical. Use a thread ID of zero to indicate the calling thread.

The *data* argument for this command must be a pointer to a *sched_aps_query_thread_parms* structure:

```
typedef struct {
    int32_t    pid;
    int32_t    tid;
    /* out parms: */
    int16_t    id;
    int16_t    inherited_id;
    uint32_t    crit_state_flags;
    int32_t    reserved1;
    int32_t    reserved2;
} sched_aps_query_thread_parms;
```

The input members include:

pid

The ID of process that the thread belongs to, or 0 to indicate the calling process.

tid

The thread ID, or 0 for the calling thread.

The output members include:

id

The ID number of the partition that the thread originally joined.

inherited_id

The ID number of the partition that the thread currently belongs to. This might not be the same as the *id* member, because the thread might have inherited the partition from a calling process.

crit_state_flags

A combination of the following flags:

- **APS_QCRIT_PERM_CRITICAL** — the thread always runs as critical.
- **APS_QCRIT_RUNNING_CRITICAL** — the thread is currently running as critical.
- **APS_QCRIT_BILL_AS_CRITICAL** — the thread's execution time is being billed to the partition's critical budget.

If **APS_QCRIT_PERM_CRITICAL** isn't set, and **APS_QCRIT_RUNNING_CRITICAL** is set, it means the thread has temporarily inherited the critical state. If **APS_QCRIT_RUNNING_CRITICAL** is set, and **APS_QCRIT_BILL_AS_CRITICAL** isn't set, it means that the thread is running as critical, but isn't depleting its partition's critical-time budget (i.e., it's running for free).

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the **SCHED_APS_QUERY_THREAD** command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

ESRCH

The specified thread wasn't found.

SCHED_APS_ATTACH_EVENTS

This command defines [sigevent](#) (p. 2961) structures that the scheduler will return to the calling thread when the scheduler detects that a given partition has become bankrupt, or the whole system has become overloaded.



Overload notification isn't implemented in this release.

Calling **SCHED_APS_ATTACH_EVENTS** arms the notification once. After you receive the notification, you must call **SCHED_APS_ATTACH_EVENTS** again to receive a subsequent notification. This is to ensure that the system doesn't send you notifications faster than you can handle them. The *pinfo_flags* field of the *sched_aps_partition_stats* structure (see the [SCHED_APS_PARTITION_STATS](#) (p. 2759) command) indicates if these events are armed.



You can register only one pair of *sigevent* structures (bankruptcy and overload) per partition, and the notifications must go to the same thread. The thread notified is the calling thread. Attaching events a second time overwrites the first. Passing NULL pointers means “no changes in notification.” To turn off notification, use *SIGEV_NONE_INIT()* to set the appropriate *sigevent* to *SIGEV_NONE*.

If you want to configure additional actions for the system to perform on bankruptcy, see “[Handling bankruptcy](#) (p. 2748),” below.

The *data* argument for this command must be a pointer to a *sched_aps_events_parm* structure:

```
typedef struct {
    const struct sigevent *bankruptcy_notification;
    const struct sigevent *overload_notification;
    /* each partition gets a different set of sigevents */
    int16_t id;
    int16_t reserved1;
    int32_t reserved2;
    int64_t reserved3;
} sched_aps_events_parm;
```

The members include:

bankruptcy_notification

A pointer to the *sigevent* to send to the calling thread if the partition becomes bankrupt, or NULL if you don't want to change the notification.

overload_notification

Not implemented.

id

The ID of the partition that you want to attach events to, or -1 for the partition of the calling thread. The command updates this member to indicate the partition that it attached the events to.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_ATTACH_EVENTS command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

You don't have the right to modify the partition, i.e., the following security modes are set and not satisfied:

- SCHED_APS_SEC_PARENT_MODIFIES
- SCHED_APS_SEC_ROOT_MAKES_PARTITIONS
- SCHED_APS_SEC_SYS_MAKES_PARTITIONS

For more information, see “[Security](#) (p. 2768),” below.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

ESRCH

The specified thread wasn't found.

SCHED_APS_ADD_SECURITY

This command sets security options. A bit that's set turns the corresponding security option on. Successive calls add to the existing set of security options. Security options can only be cleared by a restart.



You must be running in the System partition with the PROCmgr_AID_APS_ROOT ability enabled (see [procmgr_ability\(\)](#) (p. 2141)) in order to use this command, even if all security options are off.

The *data* argument for this command must be a pointer to a `sched_aps_security_parms` structure:

```
typedef struct {
    uint32_t      sec_flags;
    uint32_t      reserved1;
    uint32_t      reserved2;
} sched_aps_security_parms;
```

The members include:

sec_flags

A set of SCHED_APS_SEC_* flags (see below), as both input and output parameters. Set this member to 0 if you want to get the current security flags.

Security

The adaptive partitioning scheduler lets you dynamically create and modify the partitions in your system.

We recommend that you set up your partition environment at boot time, and then lock all parameters:



- in a program, by using the SCHED_APS_SEC_PARTITIONS_LOCKED flag
- from the command line, by using the `aps modify` command

However you might need to modify a partition at runtime. In this case, you can use the security options described below.

When QNX Neutrino starts, it sets the security option to SCHED_APS_SEC_OFF. We recommend that you immediately set it to SCHED_APS_SEC_RECOMMENDED. In code, do this:

```
sched_aps_security_parms p;

APS_INIT_DATA( &p );
p.sec_flags = SCHED_APS_SEC_RECOMMENDED;
SchedCtl( SCHED_APS_ADD_SECURITY,&p, sizeof(p) );
```



Some of the security options restrict certain operations to processes that have the PROCmgr_AID_APS_ROOT ability enabled (see [procmgr_ability\(\)](#) (p. 2141)).

The security options include the following:

SCHED_APS_SEC_RECOMMENDED

Only a process that's running in the System partition with the PROC_MGR_AID_APS_ROOT ability enabled may create partitions or change parameters. This arranges a 2-level hierarchy of partitions: the System partition and its children. Only a process that's running in the System partition with the PROC_MGR_AID_APS_ROOT ability enabled may join its own thread to partitions. The percentage budgets must not be zero.

SCHED_APS_SEC_FLEXIBLE

Only a process that's running in the System partition with the PROC_MGR_AID_APS_ROOT ability enabled can change scheduling parameters or change critical budgets. But a process that's running in *any* partition with the PROC_MGR_AID_APS_ROOT ability enabled can create subpartitions, join threads into its own subpartitions and modify subpartitions. This lets applications create their own local subpartitions out of their own budgets. The percentage budgets must not be zero.

SCHED_APS_SEC_BASIC

Only a process that's running in the System partition with the PROC_MGR_AID_APS_ROOT ability enabled may change overall scheduling parameters and set critical budgets.

Unless you're testing the partitioning and want to change all parameters without needing to restart, you should set at least SCHED_APS_SEC_BASIC.

In general, SCHED_APS_SEC_RECOMMENDED is more secure than SCHED_APS_SEC_FLEXIBLE, which is more secure than SCHED_APS_SEC_BASIC. All three allow partitions to be created and modified. After setting up partitions, use SCHED_APS_SEC_PARTITIONS_LOCKED to prevent further unauthorized changes. For example:

```
sched_aps_security_parms p;
APS_INIT_DATA( &p );
p.sec_flags = SCHED_APS_SEC_PARTITIONS_LOCKED;
SchedCtl( SCHED_APS_ADD_SECURITY, &p, sizeof(p));
```

SCHED_APS_SEC_RECOMMENDED, SCHED_APS_SEC_FLEXIBLE, and SCHED_APS_SEC_BASIC are composed of the flags defined below (but it's usually more convenient for you to use the compound options):

SCHED_APS_SEC_ROOT0_OVERALL

Your process must have the PROC_MGR_AID_APS_ROOT ability enabled and be running in the System partition in order to change the overall scheduling parameters, such as the averaging window size.

SCHED_APS_SEC_ROOT_MAKES_PARTITIONS

Your process must have the PROC_MGR_AID_APS_ROOT ability enabled in order to create or modify partitions. Applies to the [SCHED_APS_CREATE_PARTITION](#) (p. 2751), [SCHED_APS_MODIFY_PARTITION](#) (p. 2757), and [SCHED_APS_ATTACH_EVENTS](#) (p. 2766) commands.

SCHED_APS_SEC_SYS_MAKES_PARTITIONS

You must be running in the System partition in order to create or modify partitions. This applies to same commands as SCHED_APS_SEC_ROOT_MAKES_PARTITIONS. Attaching events, with SCHED_APS_ATTACH_EVENTS, is considered to be modifying the partition.

SCHED_APS_SEC_PARENT_MODIFIES

Allows partitions to be modified (SCHED_APS_MODIFY_PARTITION), but you must be running in the parent partition of the partition being modified. “Modify” means to change a partition's percentage or critical budget or attach events with the SCHED_APS_ATTACH_EVENTS command.

SCHED_APS_SEC_NONZERO_BUDGETS

A partition may not be created with, or modified to have, a zero budget. Unless you know that all your partitions need to run only in response to client requests, i.e., receipt of messages, you should set this option.

SCHED_APS_SEC_ROOT_MAKES_CRITICAL

Your process must have the PROC_MGR_AID_APS_ROOT ability enabled in order to create a nonzero critical budget or change an existing critical budget.

SCHED_APS_SEC_SYS_MAKES_CRITICAL

You must be running in the System partition to create a nonzero critical budget or change an existing critical budget.

SCHED_APS_SEC_ROOT_JOINS

Your process must have the PROC_MGR_AID_APS_ROOT ability enabled in order to join a thread to a partition.

SCHED_APS_SEC_SYS_JOINS

You must be running in the System partition in order to join a thread.

SCHED_APS_SEC_PARENT_JOINS

You must be running in the parent partition of the partition you wish to join to.

SCHED_APS_SEC_JOIN_SELF_ONLY

The caller of the [SCHED_APS_JOIN_PARTITION](#) (p. 2756) command must specify 0 for the *pid* and *tid*. In other words, a process may join only itself to a partition.

SCHED_APS_SEC_PARTITIONS_LOCKED

Prevent further changes to any partition's budget, or overall scheduling parameters, such as the window size. Set this after you've set up your partitions. Once you've locked the partitions, you can still use the [SCHED_APS_JOIN_PARTITION](#) (p. 2756) and [SCHED_APS_ATTACH_EVENTS](#) (p. 2766) commands.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the [SCHED_APS_ADD_SECURITY](#) command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EACCES

The calling thread doesn't meet the security options set (see [SCHED_APS_ADD_SECURITY](#) (p. 2767)). Your process must have the [PROC_MGR_AID_APS_ROOT](#) ability enabled.

EDOM

A reserved field isn't zero. You might not have used [APS_INIT_DATA\(\)](#) to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

SCHED_APS_QUERY_PROCESS

This command returns the partition of the given process. The partition of a process is billed while one of the process's threads handles a pulse. The individual threads in a process may all be in different partitions from the process.

The *data* argument for this command must be a pointer to a `sched_aps_query_process_parms` structure:

```
typedef struct {
    int32_t    pid;
    /* out parms: */
}
```

```
int16_t    id;        /* partition of process */
int16_t    reserved1;
int32_t    reserved2;
int32_t    reserved3;
int32_t    reserved4;
} sched_aps_query_process_parms;
```

The members include:

pid

The process ID, or 0 for the calling process.

id

The ID of the process's partition.

Errors:

SchedCtl() and *SchedCtl_r()* indicate the following errors for the SCHED_APS_QUERY_PROCESS command (see the [Returns](#) (p. 2772) section for details):

EOK

Success.

EDOM

A reserved field isn't zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.

EINVAL

The size of the parameter block doesn't match the size of the expected structure.

ENOSYS

The adaptive partitioning scheduler isn't installed.

ESRCH

The process wasn't found.

Blocking states

This call doesn't block.

Returns:

The only difference between these functions is the way they indicate errors:

SchedCtl()

EOK if the call succeeds. If an error occurs, *SchedCtl()* returns -1 and sets [errno](#) (p. 539).

SchedCtl_r()

EOK if the call succeeds. This function **doesn't** set *errno*. If an error occurs, *SchedCtl_r()* returns the negative of an error value.

For a list of error codes, see the description of each command.

Examples:

```

sched_aps_partition_info part_info;

// You need to initialize the parameter block.
APS_INIT_DATA(&part_info);

// Set the input members of the parameter block.
part_info.id = 2;

// Invoke SchedCtl to query the partition.
ret = SchedCtl( SCHED_APS_QUERY_PARTITION, &part_info,
               sizeof(part_info) );
if (EOK!=ret) some_kind_of_error_handler();

// Use output field
printf( "The budget is %d per cent.\n",
        part_info.budget_percent);

```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SchedGet(), SchedGet_r()

Get the scheduling policy for a thread

Synopsis:

```
#include <sys/neutrino.h>

int SchedGet( pid_t pid,
              int tid,
              struct sched_param *param );

int SchedGet_r( pid_t pid,
                int tid,
                struct sched_param *param );
```

Arguments:

pid

0 or a process ID; see below.

tid

0 or a thread ID; see below.

param

A pointer to a [sched_param](#) (p. 2731) structure where the function can store the scheduling parameters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SchedGet()* and *SchedGet_r()* kernel calls return the current scheduling policy and the parameters for the thread specified by *tid* in the process specified by *pid*. If *pid* is zero, the current process is used to look up a nonzero *tid*. If *pid* and *tid* are zero, then the calling thread is used.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2775) section for details.

- Instead of using these kernel calls directly, consider calling [pthread_getschedparam\(\)](#) (p. 2299).
-  In order to get the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the PROC_MGR_AID_SCHEDULE ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The scheduling policy is returned on success and is one of SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, or SCHED_OTHER.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SchedGet()

The current scheduling policy. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

SchedGet_r()

The current scheduling policy. This function does **NOT** set [errno](#). If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process indicated by *pid* or thread indicated by *tid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SchedInfo(), SchedInfo_r()

Get scheduler information

Synopsis:

```
#include <sys/neutrino.h>

int SchedInfo( pid_t pid,
               int policy,
               struct _sched_info* info );

int SchedInfo_r( pid_t pid,
                 int policy,
                 struct _sched_info* info );
```

Arguments:

pid

A process ID, or 0 to get information about the current process.

policy

One of the following:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher-priority thread.
- **SCHED_RR** — the same as **SCHED_FIFO**, except threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- **SCHED_OTHER** — currently the same as **SCHED_RR**.
- **SCHED_SPORADIC** — sporadic scheduling. For more information, see [pthread_attr_setschedpolicy\(\)](#) (p. 2223), and “Scheduling policies” in the chapter on the microkernel in the *System Architecture* guide.

info

A pointer to a `_sched_info` structure where the function can store the scheduler information.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls return information about the kernel's thread scheduler, including the minimum and maximum thread priority, for the process ID specified by *pid* when using the specified scheduling *policy*. If *pid* is 0, the scheduler information for the current process is returned. In either case, the `struct _sched_info` pointed to by *info* is filled in with the appropriate information.

The *SchedInfo()* and *SchedInfo_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2778) section for details.



Instead of using these kernel calls directly, consider calling [sched_get_priority_max\(\)](#) (p. 2725), [sched_get_priority_min\(\)](#) (p. 2727), or [sched_rr_get_interval\(\)](#) (p. 2736).

The `struct _sched_info` structure pointed to by *info* contains at least these members:

`uint64_t interval`

The current execution time limit before the thread is suspended in favor of the scheduler.

`int priority_max`

The maximum priority for a thread using this scheduling *policy*.

`int priority_min`

The minimum priority for a thread using this scheduling *policy*.

Returns:

The only difference between these functions is the way they indicate errors:

SchedInfo()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

SchedInfo_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EINVAL

The *pid* or *policy* is invalid.

ENOSYS

The *SchedInfo()* function isn't supported by this system.

ESRCH

The process specified by *pid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SchedSet(), SchedSet_r()

Set the scheduling policy for a thread

Synopsis:

```
#include <sys/neutrino.h>

int SchedSet(
    pid_t pid,
    int tid,
    int policy,
    const struct sched_param *param );

int SchedSet_r(
    pid_t pid,
    int tid,
    int policy,
    const struct sched_param *param );
```

Arguments:

pid

0 or a process ID; see below.

tid

0 or a thread ID; see below.

policy

The scheduling policy; one of:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher priority thread.
- **SCHED_RR** — the same as **SCHED_FIFO**, except threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- **SCHED_OTHER** — currently the same as **SCHED_RR**.
- **SCHED_SPORADIC** — sporadic scheduling.
- **SCHED_NOCHANGE** — this isn't actually a policy, but a special value that tells the kernel to update the parameters specified in *param*, without changing the policy.
- **SCHED_ADJTOHEAD** — if the thread specified by *pid* and *tid* is **READY**, put it at the head of the **READY** queue.

- **SCHED_ADJTOTAIL** — if the thread specified by *pid* and *tid* is READY, put it at the tail of the READY queue.



If you specify **SCHED_ADJTOHEAD** or **SCHED_ADJTOTAIL**, the *param* argument is ignored.

For more information, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

param

A pointer to a [sched_param](#) (p. 2731) structure that holds the new scheduling parameters.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SchedSet()* and *SchedSet_r()* kernel calls set both the scheduling policy and the associated parameters for the thread specified by *tid* in the process specified by *pid*. If *pid* is zero, the current process is used to look up a nonzero *tid*. If *tid* is zero, then the calling thread is used and *pid* is ignored.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2782) section for details.

- Instead of using these kernel calls directly, consider calling [pthread_setschedparam\(\)](#) (p. 2400) or [pthread_setschedprio\(\)](#) (p. 2402).



- In order to set the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the **PROCmgr_AID_SCHEDULE** ability enabled. In order to set the priority above the maximum allowed for unprivileged processes, your process must have the **PROCmgr_AID_PRIORITY** ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
-

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SchedSet()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

SchedSet_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers you provided.

EINVAL

The given scheduling *policy* is invalid.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process indicated by *pid* or thread indicated by *tid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SchedYield(), SchedYield_r()

Yield to other threads

Synopsis:

```
#include <sys/neutrino.h>

int SchedYield( void );

int SchedYield_r( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls check to see if other threads at the same priority as that of the calling thread are ready to run. If so, the calling thread yields to them and places itself at the end of the READY thread queue for that priority. *SchedYield()* never yields to a lower-priority thread. Higher-priority threads always force a yield the instant they become ready to run. This call has no effect with respect to threads running at priorities other than the calling thread's.

The *SchedYield()* and *SchedYield_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2784) section for details.



Instead of using these kernel calls directly, consider calling [sched_yield\(\)](#) (p. 2743).

Avoid designing programs that contain busy-wait loops using *SchedYield()* to timeslice. If this is unavoidable, you can use *SchedYield()* to reduce the system load at a given priority level. Note that a program that calls *SchedYield()* in a tight loop will spend a great deal of time in the kernel, which will have a small effect on scheduling interrupt latency.

Blocking states

These calls don't block. However, if other threads are ready at the same priority, the calling thread is placed at the end of the ready queue for this priority.

Returns:

The only difference between these functions is the way they indicate errors:

SchedYield()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

SchedYield_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

searchenv()

Search the directories listed in an environment variable

Synopsis:

```
#include <stdlib.h>

void searchenv( const char* name,
                const char* env_var,
                char* buffer );
```

Arguments:

name

The name of the file that you want to search for.

env_var

The name of an environment variable whose value is a list of directories that you want to search. Common values for *env_var* are "PATH", "LIB" and "INCLUDE".



The *searchenv()* function doesn't search the current directory unless it's specified in the environment variable.

buffer

A buffer where the function can store the full path of the file found. This buffer should be PATH_MAX bytes long. If the specified file can't be found, the function stores an empty string in the buffer.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *searchenv()* function searches for the file specified by *name* in the list of directories assigned to the environment variable specified by *env_var*.



Use [pathfind\(\)](#) (p. 1939) or [pathfind_r\(\)](#) (p. 1939) instead of this function.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void display_help( FILE *fp )
{
    printf( "display_help T.B.I.\n" );
}

int main( void )
{
    FILE *help_file;
    char full_path[ PATH_MAX ];

    searchenv( "lib_ref.html", "PATH", full_path );
    if( full_path[0] == '\0' ) {
        printf( "Unable to find help file\n" );
    } else {
        help_file = fopen( full_path, "r" );
        display_help( help_file );
        fclose( help_file );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The `searchenv()` function manipulates the environment pointed to by the global [environ](#) (p. 528) variable.

seed48()

Initialize the seed for a sequence of pseudo-random numbers

Synopsis:

```
#include <stdlib.h>

unsigned short int *seed48(
    unsigned short int seed16v[3] );
```

Arguments:

seed16v

An array that comprises the 48 bits of the seed.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `seed48()` initializes the internal buffer $r(n)$ of `drand48()`, `lrand48()`, and `mrnd48()`. All 48 bits of the seed can be specified in an array of 3 short integers, where the entry with index 0 specifies the lowest bits. The constant multiplicand and addend of the algorithm are reset to the defaults: the multiplicand $a = 0xFDEECE66D = 25214903917$ and the addend $c = 0xB = 11$.

Returns:

A pointer to an array of 3 shorts which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to `seed48()`.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

seekdir()

Set the position for the next read of the directory stream

Synopsis:

```
#include <dirent.h>

void seekdir( DIR * dirp,
              long int pos );
```

Arguments:

dirp

A pointer to the directory stream, for which you want to set the current location.

pos

The new position for the directory stream, which you must have obtained from an earlier call to [telldir\(\)](#) (p. 3375).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *seekdir()* function sets the position of the next [readdir\(\)](#) (p. 2514) operation on the directory stream specified by *dirp* to the position specified by *pos*.

The new position reverts to the one associated with the directory stream when the *telldir()* operation was performed.

Values returned by *telldir()* are good only for the lifetime of the `DIR` pointer, *dirp*, from which they're derived. If you call [rewinddir\(\)](#) (p. 2648) on the directory stream, or if you close and reopen the directory, an offset previously returned by *telldir()* becomes invalid.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

select()

Check for files that are ready for reading or writing

Synopsis:

```
#include <sys/select.h>

int select( int width,
            fd_set * readfds,
            fd_set * writefds,
            fd_set * exceptfds,
            struct timeval * timeout );

FD_SET( int fd, fd_set * fdset );
FD_CLR( int fd, fd_set * fdset );
FD_ISSET( int fd, fd_set * fdset );
FD_ZERO( fd_set * fdset );
```

Arguments:***width***

The number of descriptors to check in the given sets. Only the descriptors from 0 through (*width*-1) in the descriptor sets are examined. Therefore, the value of *width* must be at least as large as:

(highest valued file descriptor in the sets) +1

readfds

NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for reading. The function replaces the set with the file descriptors that are actually ready for reading.

writefds

NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for writing. The function replaces the set with the file descriptors that are actually ready for writing.

exceptfds

NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that have an exceptional condition pending. The function replaces the set with the file descriptors that actually have an exceptional condition pending.

timeout

NULL, or a pointer to a `timeval` that specifies how long to wait for the selection to complete.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `select()` function examines the file descriptor sets whose addresses are passed in `readfds`, `writfds`, and `exceptfds` to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending. Any of `readfds`, `writfds`, and `exceptfds` may be NULL pointers if no descriptors are of interest.



Until QNX Neutrino 6.1, `select()` and the associated macros were defined in `sys/time.h`. They're now defined in `sys/select.h`, which `sys/time.h` includes.

The `select()` function replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation, and returns the total number of ready descriptors in all the sets.

If `timeout` isn't NULL, it specifies a maximum interval to wait for the selection to complete. If `timeout` is NULL, `select()` blocks until one of the selected conditions occurs. To effect a poll, the `timeout` argument should be a non-NULL pointer, pointing to a zero-valued `timeval` structure.



Because of the nature of time measurement, the function might actually wait longer than the specified time. Polling can also interfere with the kernel's efforts to manage power usage. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

If the current operating system configuration supports a larger number of open files than is specified in `FD_SETSIZE`, you can increase the number of open file descriptors used with `select()` by changing the definition of `FD_SETSIZE` before including `<sys/select.h>` or `<sys/time.h>`.

If you use `select()` with a timeout, you should reset the timeout value after calling `select()`.



If you're using `select()` in conjunction with the socket API package, note that selecting for reading on a socket descriptor on which a [listen\(\)](#) (p. 1523) has

been performed indicates that a subsequent [accept\(\)](#) (p. 58) on that descriptor won't block.

Manipulating file-descriptor sets

At least the following macros are defined in `<sys/select.h>` for manipulating file-descriptor sets:

FD_ZERO(&fdset)

Initialize a descriptor set *fdset* to the null set.

FD_SET(fd, &fdset)

Add the file descriptor *fd* to the set *fdset*.

FD_CLR(fd, &fdset)

Remove *fd* from *fdset*.

FD_ISSET(fd, &fdset)

Is nonzero if *fd* is a member of *fdset*; otherwise, zero.

The behavior of these macros is undefined if a descriptor value is less than zero, or greater than or equal to `FD_SETSIZE`.

Returns:

The number of ready descriptors in the descriptor sets, 0 if the *timeout* expired, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

One of the descriptor sets specified an invalid descriptor.

EFAULT

One of the pointers given in the call referred to a nonexistent portion of the address space for the process.

EINTR

A signal was delivered before any of the selected events occurred, or before the time limit expired.

EINVAL

A component of the pointed-to time limit is outside the acceptable range:
 t_sec must be between 0 and 10^8 , inclusive; t_usec must be greater than or equal to 0, and less than 10^6 .

Examples:

```

/*
 * This example opens a console and a serial port for
 * read mode, and calls select() with a 5 second timeout.
 * It waits for data to be available on either descriptor.
 */

#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/select.h>

int main( void )
{
    int console, serial;
    struct timeval tv;
    fd_set rfd;
    int n;

    if( ( console = open( "/dev/con1", O_RDONLY ) ) == -1
    ||   ( serial = open( "/dev/ser1", O_RDONLY ) ) == -1 )
    {
        perror( "open" );
        return EXIT_FAILURE;
    }

    /*
     * Clear the set of read file descriptors, and
     * add the two we just got from the open calls.
     */
    FD_ZERO( &rfd );
    FD_SET( console, &rfd );
    FD_SET( serial, &rfd );

    /*
     * Set a 5 second timeout.
     */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    switch ( n = select( 1 + max( console, serial ),
                        &rfd, 0, 0, &tv ) ) {
        case -1:
            perror( "select" );
            return EXIT_FAILURE;
        case 0:
            puts( "select timed out" );
            break;
        default:
            printf( "%d descriptors ready ...\n", n );
            if( FD_ISSET( console, &rfd ) )
                puts( " -- console descriptor has data pending" );
            if( FD_ISSET( serial, &rfd ) )
                puts( " -- serial descriptor has data pending" );
    }
    return EXIT_SUCCESS;
}

```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Read the <i>Caveats</i>

Caveats:

The *select()* function works only with raw file descriptors; it doesn't work with file descriptors in edited mode. See the `ICANON` flag in the description of the [tcgetattr\(\)](#) (p. 3347) function.

The *select()* function is thread safe as long as the *fd* sets used by each thread point to memory that is specific to that thread.

In QNX Neutrino, if multiple threads block in *select()* on the same *fd* for the same condition, all threads may unblock when the condition is satisfied. This may differ from other implementations where only one thread may unblock.

select_attach()

Attach a file descriptor to a dispatch handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int select_attach
( void *dpp,
  select_attr_t *attr,
  int fd,
  unsigned flags,
  int (*func)( select_context_t *ctp,
               int fd,
               unsigned flags,
               void *handle ),
  void *handle );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433), that you want to attach to a file descriptor.

attr

A pointer to a `select_attr_t` structure. This structure is defined as:

```
typedef struct _select_attr {
    unsigned flags;
} select_attr_t;
```

Currently, no attribute flags are defined.

fd

The file descriptor that you want to attach to the dispatch handle.

flags

Flags that specify the events that you're interested in. For more information, see “[Flags](#) (p. 2797),” below.

func

The function that you want to call when the file descriptor unblocks. For more information, see “[Function](#) (p. 2798),” below.

handle

A pointer to arbitrary data that you want to pass to *func*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *select_attach()* attaches the file descriptor *fd* to the dispatch handle *dpp* and selects *flags* (p. 2797) events. When *fd* “unblocks”, *func* is called with *handle*.

Flags

The available flags are defined in `<sys/dispatch.h>`. The following flags use *ionotify()* mechanisms (see *ionotify()* (p. 1386) for further details):

SELECT_FLAG_EXCEPT

Out-of-band data is available. The definition of out-of-band data depends on the resource manager.

SELECT_FLAG_READ

There's input data available. The amount of data available defaults to 1. For a character device such as a serial port, this is a character. For a POSIX message queue, it's a message. Each resource manager selects an appropriate object.

SELECT_FLAG_WRITE

There's room in the output buffer for more data. The amount of room available needed to satisfy this condition depends on the resource manager. Some resource managers may default to an empty output buffer, while others may choose some percentage of the empty buffer.

These flags are specific to dispatch:

SELECT_FLAG_REARM

Rearm the *fd* after an event is dispatched.

SELECT_FLAG_SRVEXCEPT

Register a function that's called whenever a server, to which this client is connected, dies. (This flag uses the *ChannelCreate()* (p. 278) function's `_NTO_CHF_COID_DISCONNECT` flag. In this case, *fd* is ignored.)

Function

The argument *func* is the user-supplied function that's called when one of the registered events occurs on *fd*. This function should return 0 (zero); other values are reserved. The function is passed the following arguments:

ctp

Context pointer.

fd

The *fd* on which the event occurred.

flags

The type of event that occurred. The possible *flags* are:

- `SELECT_FLAG_EXCEPT`
- `SELECT_FLAG_READ`
- `SELECT_FLAG_WRITE`

For descriptions of the flags passed to *func*, see “[Flags](#) (p. 2797),” above.

handle

The *handle* passed to *select_attach()*.

Returns:

Zero on success, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EINVAL**

Invalid argument.

ENOMEM

Insufficient memory was available.

Examples:

For an example with *select_attach()*, see [dispatch_create\(\)](#) (p. 433). For other examples using the dispatch interface, see [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

select_detach()

Detach a file descriptor from a dispatch handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int select_detach( void *dpp,
                  int fd );
```

Arguments:

dpp

The dispatch handle, as returned by a successful call to [dispatch_create\(\)](#) (p. 433), that you want to detach from the file descriptor.

fd

The file descriptor that you want to detach.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function *select_detach()* detaches the file descriptor *fd* that was registered with dispatch *dpp*, using the [select_attach\(\)](#) (p. 2796) call.

Returns:

0

Success.

-1

The file descriptor *fd* wasn't registered with the dispatch *dpp*.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>
```

```

#include <fcntl.h>

int my_func( ... ) {
    ...
}

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    int             fd;
    select_attr_t    attr;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
            dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    if( argc < 2 || ( fd = open( argv[1],
        O_RDWR | O_NONBLOCK )) == -1 ) {
        exit(0);
    }

    select_attach( dpp, &attr, fd,
        SELECT_FLAG_READ | SELECT_FLAG_REARM, my_func, NULL );

    ...

    if ( (select_detach( dpp, fd )) == -1 ) {
        fprintf( stderr, "Failed to detach \
            the file descriptor.\n" );
        return 1;
    }
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

select_query()

Decode the last select event

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int select_query
( select_context_t *ctp,
  int *fd,
  unsigned *flags,
  int (**func)( select_context_t *ctp,
                int fd,
                unsigned flags,
                void *handle ),
  void **handle );
```

Arguments:

ctp

A pointer to a `select_context_t` structure that defines the context of the event that you want to get information about.

fd

A pointer to a location where the function can store the file descriptor that's associated with the event.

flags

A pointer to a location where the function can store the flags associated with the event; see “[Flags](#) (p. 2797)” in the documentation for `select_attach()`.

func

A pointer to a location where the function can store the function associated with the event; see “[Function](#) (p. 2798)” in the documentation for `select_attach()`.

handle

A pointer to a location where the function can store the address of any data that you arranged to pass to `func`.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The function `select_query()` stores the values of the last select event, for context `ctp`, in `fd`, `flags`, `func`, and `handle`.

Returns:

If an error occurs, the function returns -1. An error occurs if the received event doesn't belong to one of the file descriptors attached with [select_attach\(\)](#) (p. 2796).

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int my_func( select_context_t *ctp,
            int fd,
            unsigned flags,
            void *handle ) {
    ...
}

int main( int argc, char **argv ) {
    dispatch_t      *dpp;
    dispatch_context_t *ctp;
    int             fd;
    unsigned        flag;
    void            *handle;
    select_attr_t    *attr;
    int             (*func)( select_context_t *,
                             int, unsigned, void * );

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    if( argc < 2 || (fd = open( argv[1],
                               O_RDWR | O_NONBLOCK )) == -1 ) {
        exit(0);
    }

    select_attach( dpp, attr, fd,
                  SELECT_FLAG_READ | SELECT_FLAG_REARM, &my_func, NULL );

    ctp = dispatch_context_alloc( dpp );

    ...

    if( select_query( (select_context_t *)ctp, &fd, &flag,
                     &func, &handle ) == -1 ) {
        fprintf( stderr, "Failed to decode last select event.\n");
        return 1;
    }
}
```

For more examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

sem_close()

Close a named semaphore

Synopsis:

```
#include <semaphore.h>

int sem_close( sem_t * sem );
```

Arguments:

sem

A pointer to a semaphore, as returned by [sem_open\(\)](#) (p. 2814).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_close()` function closes the named semaphore `sem` opened by `sem_open()`, releasing any system resources associated with the `sem`.



Don't mix named semaphore operations (`sem_open()` and `sem_close()`) with unnamed semaphore operations (`sem_init()` (p. 2811) and `sem_destroy()` (p. 2807)) on the same semaphore.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

Invalid semaphore descriptor `sem`.

Classification:

POSIX 1003.1 SEM (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_destroy()

Destroy an unnamed semaphore

Synopsis:

```
#include <semaphore.h>

int sem_destroy( sem_t * sem );
```

Arguments:

sem

A pointer to the `sem_t` object for the semaphore that you want to destroy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_destroy()` function destroys the unnamed semaphore referred to by the `sem` argument. The semaphore must have been previously initialized by the [sem_init\(\)](#) (p. 2811) function.

The effect of using a semaphore after it has been destroyed is undefined. If you destroy a semaphore that other processes are currently blocked on, they're unblocked, with an error (EINVAL).



Don't mix named semaphore operations ([sem_open\(\)](#) (p. 2814) and [sem_close\(\)](#) (p. 2805)) with unnamed semaphore operations (`sem_init()` and `sem_destroy()`) on the same semaphore.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EINVAL**

Invalid semaphore descriptor *sem*.

Classification:

[*POSIX 1003.1 SEM*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_getvalue()

Get the value of a named or unnamed semaphore

Synopsis:

```
#include <semaphore.h>

int sem_getvalue( sem_t * sem,
                  int * value );
```

Arguments:

sem

A pointer to the `sem_t` object for the semaphore whose value you want to get.

value

A pointer to a location where the function can store the semaphore's value. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a *value* of 0 (zero) indicates a locked semaphore.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_getvalue()` function takes a snapshot of the value of the semaphore, *sem*, and stores it in *value*. This value can change at any time, and is guaranteed valid only at some point in the `sem_getvalue()` call.

This function is provided for debugging semaphore code.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EINVAL**

Invalid semaphore descriptor *sem*.

Classification:

POSIX 1003.1 SEM (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_init()

Initialize an unnamed semaphore

Synopsis:

```
#include <semaphore.h>

int sem_init( sem_t * sem,
              int pshared,
              unsigned value );
```

Arguments:

sem

A pointer to the `sem_t` object for the semaphore that you want to initialize.

pshared

Nonzero if you want the semaphore to be shared between processes via shared memory.

value

The initial value of the semaphore. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a *value* of 0 (zero) indicates a locked semaphore. This value must not exceed `SEM_VALUE_MAX`.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `sem_init()` function initializes the unnamed semaphore referred to by the *sem* argument. The initial counter value of this semaphore is specified by the *value* argument.



You should allocate synchronization objects only in normal memory mappings. On certain processors, atomic operations such as calls to [pthread_mutex_lock\(\)](#) (p. 2318) will cause a fault if the control structure is allocated in uncached memory.

You can use the initialized semaphore in subsequent calls to [sem_wait\(\)](#) (p. 2827), [sem_trywait\(\)](#) (p. 2823), [sem_post\(\)](#) (p. 2818), and [sem_destroy\(\)](#) (p. 2807). An initialized semaphore is valid until it's destroyed by the [sem_destroy\(\)](#) function, or until the memory where the semaphore resides is released.

If the *pshared* argument is nonzero, then the semaphore can be shared between processes via shared memory. Any process can then use *sem* with the [sem_wait\(\)](#), [sem_trywait\(\)](#), [sem_post\(\)](#) and [sem_destroy\(\)](#) functions.



Don't mix named semaphore operations ([sem_open\(\)](#) (p. 2814) and [sem_close\(\)](#) (p. 2805)) with unnamed semaphore operations ([sem_init\(\)](#) and [sem_destroy\(\)](#)) on the same semaphore.

Returns:

0

Success. The semaphore referred to by *sem* is initialized.

-1

An error occurred (*errno* is set).

Errors:

EAGAIN

A resource required to initialize the semaphore has been exhausted.

EBUSY

The given semaphore was previously initialized, and has not been destroyed.

EINVAL

The *value* argument exceeds SEM_VALUE_MAX.

EPERM

The process lacks the appropriate privileges to initialize the semaphore.

ENOSPC

A resource required to initialize the semaphore has been exhausted.

ENOSYS

The [sem_init\(\)](#) function isn't supported.

Classification:

[POSIX 1003.1 SEM](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

Don't initialize the same semaphore from more than one thread. It's best to set up semaphores before starting any threads.

sem_open()

Create or access a named semaphore

Synopsis:

```
#include <semaphore.h>
#include <fcntl.h>

sem_t * sem_open( const char * sem_name,
                  int oflags,
                  ... );
```

Arguments:

sem_name

The name of the semaphore that you want to create or access; see below.

oflags

Flags that affect how the function creates a new semaphore. This argument is a combination of:

- **O_CREAT**
- **O_EXCL**



Don't set *oflags* to **O_RDONLY**, **O_RDWR**, or **O_WRONLY**. A semaphore's behavior is undefined with these flags. The QNX Neutrino libraries silently ignore these options, but they may reduce your code's portability.

For more information, see below.

If you set **O_CREAT** in *oflags*, you must also pass the following arguments:

mode_t mode

The semaphore's mode (just like file modes). For portability, you should set the read, write, *and* execute bits to the same value. An easy way of doing this is to use the constants from `<sys/stat.h>`:

- **S_IRWXG** for group access.
- **S_IRWXO** for other's access.
- **S_IRWXU** for your own access.

For more information, see “[Access permissions](#) (p. 3165)” in the documentation for `stat()`.

unsigned int *value*

The initial value of the semaphore. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a *value* of 0 (zero) indicates a locked semaphore. This value must not exceed `SEM_VALUE_MAX`.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `sem_open()` function creates or accesses a named semaphore. Named semaphores are slower than the unnamed semaphores created with `sem_init()` (p. 2811). Semaphores persist as long as the system is up.



If you want to use named semaphores, the named-semaphore manager must be running. Starting with release 6.3.0, `procnto` manages named semaphores directly, which `mqueue` used to do (and still does, if it detects that `procnto` isn't doing so).

The `sem_open()` function returns a semaphore descriptor that you can use with `sem_wait()` (p. 2827), `sem_trywait()` (p. 2823), and `sem_post()` (p. 2818). You can use it until you call `sem_close()` (p. 2805). You can unlink the semaphore by calling `sem_unlink()` (p. 2825); when all processes have unlinked the semaphore, it's destroyed.

Named semaphores are always created under `/dev/sem`:

- If the *name* argument starts with a slash, the semaphore is given that name.
- If the *name* argument doesn't begin with a slash character, the semaphore is given that name, prepended with the current working directory.

In either case, slash characters other than the leading slash character aren't interpreted, and the specified name, including these slash characters, is used to identify the semaphore. In other words, additional slashes don't create a directory structure under `/dev/sem`.

For example, if your current directory is `/tmp`:

<i>name</i>	Pathname space entry
<code>/my_sem</code>	<code>/dev/sem/my_sem</code>

<i>name</i>	Pathname space entry
my_sem	/dev/sem/tmp/my_sem



If you want to create or access a semaphore on another node, specify the name as `/net/node/sem_location`.

The *oflags* argument is used only for semaphore creation. When creating a new semaphore, you can set *oflags* to `O_CREAT` or `(O_CREAT|O_EXCL)`:

O_CREAT

Create a new named semaphore. If you set this bit, you must provide the *mode* and *value* arguments to *sem_open()*.

O_EXCL

When creating a new named semaphore, `O_EXCL` causes *sem_open()* to fail if a semaphore with *sem_name* already exists. Without `O_EXCL`, *sem_open()* attaches to an existing semaphore or creates a new one if *sem_name* doesn't exist.



Don't mix named semaphore operations (*sem_open()* and *sem_close()*) with unnamed semaphore operations (*sem_init()* and *sem_destroy()* (p. 2807)) on the same semaphore.

Returns:

A pointer to the created or accessed semaphore, or `SEM_FAILED` for failure (*errno* (p. 539) is set).

Errors:

EACCES

Either the named semaphore exists and you don't have permission to access it, or you're trying to create a new semaphore and you don't have permission.

EEXIST

You specified `O_CREAT` and `O_EXCL` in *oflags*, but the semaphore already exists.

EINVAL

The *sem_name* argument is invalid or, when creating a semaphore, *value* is greater than `SEM_VALUE_MAX`.

EINTR

The call was interrupted by a signal.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

The process is using too many files or semaphores.

ENFILE

The system ran out of resources and couldn't open the semaphore.

ENAMETOOLONG

The *sem_name* argument is longer than (NAME_MAX - 8).

ENOENT

Either the manager for named semaphores (mqueue, if procnto isn't managing them) isn't running, or the *sem_name* argument doesn't exist and you didn't specify O_CREAT in *oflags*.

ENOSPC

There's insufficient space to create a new named semaphore.

ENOSYS

The *sem_open()* function isn't implemented for the filesystem specified in *sem_name*.

Classification:

[POSIX 1003.1 SEM](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_post()

Increment a named or unnamed semaphore

Synopsis:

```
#include <semaphore.h>

int sem_post( sem_t * sem );
```

Arguments:

sem

A pointer to the `sem_t` object for the semaphore whose value you want to increment.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `sem_post()` function increments the semaphore referenced by the `sem` argument. If any processes are currently blocked waiting for the semaphore, then one of these processes will return successfully from its call to `sem_wait`.

The process to be unblocked is determined in accordance with the scheduling policies in effect for the blocked processes. The highest priority waiting process is unblocked, and if there is more than one highest priority process blocked waiting for the semaphore, then the highest priority process that has been waiting the longest is unblocked.

The `sem_post()` function is reentrant with respect to signals, and can be called from a signal handler.

Returns:

0

Success.

-1

An error occurred ([*errno*](#) (p. 539) is set).

Errors:**EINTR**

The call was interrupted by a signal.

EINVAL

The *sem* argument doesn't point to a valid semaphore descriptor.

ENOSYS

The *sem_post()* function isn't supported.

Classification:

[POSIX 1003.1 SEM](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_timedwait(), sem_timedwait_monotonic()

Wait on a named or unnamed semaphore, with a timeout

Synopsis:

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(
    sem_t * sem,
    const struct timespec * abs_timeout );

int sem_timedwait_monotonic(
    sem_t * sem,
    const struct timespec * abs_timeout );
```

Arguments:

sem

The semaphore that you want to wait on.

abs_timeout

A pointer to a [timespec](#) (p. 3473) structure that specifies the maximum time to wait to lock the semaphore, expressed as an absolute time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_timedwait()` function locks the semaphore referenced by `sem` as in the `sem_wait()` function. However, if the semaphore can't be locked without waiting for another process or thread to unlock the semaphore by calling [sem_post\(\)](#) (p. 2818), the wait is terminated when the specified timeout expires. The timeout is based on the `CLOCK_REALTIME` clock.

The `sem_timedwait_monotonic()` function is a QNX Neutrino extension; it's similar to `sem_timedwait()`, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

The timeout expires when the absolute time specified by `abs_timeout` passes, as measured by the clock on which timeouts are based (i.e. when the value of that clock

equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

Returns:

0

The calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*.

-1

The call was unsuccessful (*errno* (p. 539) is set). The state of the semaphore is unchanged.

Errors:**EDEADLK**

A deadlock condition was detected.

EINTR

A signal interrupted this function.

EINVAL

Invalid semaphore *sem*, or the thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

ETIMEDOUT

The semaphore couldn't be locked before the specified timeout expired.

Examples:

```
#include <stdio.h>
#include <semaphore.h>
#include <time.h>

main(){
    struct timespec tm;
    sem_t sem;
    int i=0;

    sem_init( &sem, 0, 0);

    do {
        clock_gettime(CLOCK_REALTIME, &tm);
        tm.tv_sec += 1;
        i++;
        printf("i=%d\n",i);
        if (i==10) {
            sem_post(&sem);
        }
    } while ( sem_timedwait( &sem, &tm ) == -1 );
}
```

```
    printf("Semaphore acquired after %d timeouts\n", i);  
    return;  
}
```

Classification:

`sem_timedwait()` is [POSIX 1003.1 SEM TMO](#) (p. 46); `sem_timedwait_monotonic()` is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_trywait()

Wait on a named or unnamed semaphore, but don't block

Synopsis:

```
#include <semaphore.h>

int sem_trywait( sem_t * sem );
```

Arguments:

sem

A pointer to the `sem_t` object for the semaphore that you want to wait on.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_trywait()` function decrements the semaphore if the semaphore's value is greater than zero; otherwise, the function simply returns.

Returns:

0

The semaphore was successfully decremented.

-1

The state of the semaphore is unchanged ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The semaphore was already locked, so it couldn't be immediately locked by the `sem_trywait()` function.

EDEADLK

A deadlock condition was detected.

EINVAL

Invalid semaphore descriptor *sem*.

EINTR

A signal interrupted this function.

Classification:

[*POSIX 1003.1 SEM*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_unlink()

Destroy a named semaphore

Synopsis:

```
#include <semaphore.h>

int sem_unlink( const char * sem_name );
```

Arguments:

sem_name

The name of the semaphore that you want to destroy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_unlink()` function destroys the named semaphore, `sem_name`. Open semaphores are removed the same way that [unlink\(\)](#) (p. 3579) removes open files; the processes that have the semaphore open can still use it, but the semaphore will disappear as soon as the last process uses `sem_close()` to close it.

Any attempt to use [sem_open\(\)](#) (p. 2814) on an unlinked semaphore will refer to a *new* semaphore.

Semaphores are persistent as long as the system remains up.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

You don't have permission to unlink the semaphore.

ELOOP

Too many levels of symbolic links or prefixes.

ENOENT

The semaphore *sem_name* doesn't exist.

ENAMETOOLONG

The *sem_name* argument is longer than (NAME_MAX - 8).

ENOSYS

The *sem_unlink()* function isn't implemented for the filesystem specified in *path*.

Classification:

[*POSIX 1003.1 SEM*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sem_wait()

Wait on a named or unnamed semaphore

Synopsis:

```
#include <semaphore.h>

int sem_wait( sem_t * sem );
```

Arguments:

sem

A pointer to the `sem_t` object for the semaphore that you want to wait on.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_wait()` function decrements the semaphore referred to by the `sem` argument. If the semaphore value is not greater than zero, then the calling process blocks until it can decrement the counter, or the call is interrupted by signal.

Some process should eventually call [sem_post\(\)](#) (p. 2818) to increment the semaphore.

Returns:

0

The semaphore was successfully decremented.

-1

The state of the semaphore is unchanged ([errno](#) (p. 539) is set).

Errors:

EDEADLK

A deadlock condition was detected.

EINVAL

Invalid semaphore descriptor *sem*.

EINTR

A signal interrupted this function.

Classification:

[*POSIX 1003.1 SEM*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

send()

Send a message to a connected socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send( int s,
              const void * msg,
              size_t len,
              int flags );
```

Arguments:

s

The descriptor for the socket; see [socket\(\)](#) (p. 3075).

msg

A pointer to the message that you want to send.

len

The length of the message.

flags

A combination of the following:

- **MSG_OOB** — process out-of-band data. Use this bit when you send “out-of-band” data on sockets that support this notion (e.g. **SOCK_STREAM**). The underlying protocol must also support out-of-band data.
- **MSG_DONTROUTE** — bypass routing; create a direct interface. You normally use this bit only in diagnostic or routing programs.
- **MSG_NOSIGNAL** — don't raise a **SIGPIPE** signal when the other end breaks the connection.

Library:

libsocket

Use the **-l socket** option to **qcc** to link against this library.

Description:

The `send()`, `sendto()` (p. 2837), and `sendmsg()` (p. 2833) functions are used to transmit a message to another socket. The `send()` function can be used only when the socket is in a *connected* state, while `sendto()` and `sendmsg()` can be used at any time.

The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error `EMSGSIZE` is returned, and the message isn't transmitted.

No indication of failure to deliver is implicit in a `send()`. Locally detected errors are indicated by a return value of -1.

If no message space is available at the socket to hold the message to be transmitted, then `send()` normally blocks, unless the socket has been placed in nonblocking I/O mode. You can use `select()` (p. 2791) to determine when it's possible to send more data.

Returns:

The number of bytes sent, or -1 if an error occurs (`errno` (p. 539) is set).

Errors:**EACCES**

The calling process doesn't have the appropriate privileges.

EAGAIN

The socket's file descriptor is marked `O_NONBLOCK`, and the requested operation would block.

EBADF

An invalid descriptor was specified.

ECONNRESET

A connection was forcibly closed by a peer.

EDESTADDRREQ

The socket is not connection-mode and no peer address is set.

EFAULT

An invalid user space address was specified for a parameter.

EINTR

A signal interrupted `send()` before any data was transmitted.

EIO

An I/O error occurred while reading from or writing to the filesystem.

EMSGSIZE

The socket requires that the message be sent atomically, but the size of the message made this impossible.

ENETDOWN

The local network interface used to reach the destination is down.

ENETUNREACH

No route to the network is present.

ENOBUFS

The system couldn't allocate an internal buffer. The operation may succeed when buffers become available.

ENOTCONN

The socket isn't connected or otherwise has not had the peer pre-specified.

ENOTSOCK

The argument *s* isn't a socket.

EOPNOTSUPP

The socket argument is associated with a socket that doesn't support one or more of the values set in *flags*.

EPIPE

The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type `SOCK_STREAM`, a `SIGPIPE` signal is generated to the calling thread.

EWouldBlock

The socket is marked nonblocking and the requested operation would block.

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

sendmsg()

Send a message and its header to a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendmsg( int s,
                 const struct msghdr * msg,
                 int flags );
```

Arguments:

s

The descriptor for the socket; see [socket\(\)](#) (p. 3075).

msg

A pointer to the message that you want to send. For a description of the msghdr structure, see [recvmsg\(\)](#) (p. 2539).

flags

A combination of the following:

- MSG_OOB — process out-of-band data. Use this bit when you send “out-of-band” data on sockets that support this notion (e.g. SOCK_STREAM). The underlying protocol must also support out-of-band data.
- MSG_DONTROUTE — bypass routing; create a direct interface. You normally use this bit only in diagnostic or routing programs.
- MSG_NOSIGNAL — don't raise a SIGPIPE signal when the other end breaks the connection.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The `sendmsg()` function is used to transmit a message to another socket. You can use [send\(\)](#) (p. 2829) only when the socket is in a *connected* state; you can use `sendmsg()` at any time.

No indication of failure to deliver is implicit in a `sendmsg()`. Locally detected errors are indicated by a return value of -1.

If no message space is available at the socket to hold the message to be transmitted, then `sendmsg()` normally blocks, unless the socket has been placed in nonblocking I/O mode. You can use [select\(\)](#) (p. 2791) to determine when it's possible to send more data.

Returns:

The number of bytes sent, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied for a component of the path prefix, or write access to the named socket is denied.

EAGAIN

The socket's file descriptor is marked `O_NONBLOCK`, and the requested operation would block.

EAFNOSUPPORT

Addresses in the specified address family cannot be used with this socket.

EBADF

An invalid descriptor was specified.

ECONNRESET

A connection was forcibly closed by a peer.

EDESTADDRREQ

The socket isn't connection-mode and doesn't have its peer address set, and no destination address was specified.

EFAULT

An invalid user space address was specified for a parameter.

EHOSTUNREACH

The destination host can't be reached (probably because the host is down, or a remote router can't reach it).

EINTR

A signal interrupted *sendmsg()* before any data was transmitted.

EINVAL

The sum of the *iov_len* values overflows an *ssize_t*.

EIO

An I/O error occurred while reading from or writing to the filesystem.

EISCONN

A destination address was specified and the socket is already connected.

EMSGSIZE

The message is too large to be sent all at once (as the socket requires), or the *msg_iovlen* member of the *msghdr* structure pointed to by *message* is less than or equal to 0 or is greater than *IOV_MAX*.

ENETDOWN

The local network interface used to reach the destination is down.

ENETUNREACH

No route to the network is present.

ENOBUFS

The system couldn't allocate an internal buffer. The operation may succeed when buffers become available.

ENOMEM

Insufficient memory was available to fulfill the request.

ENOTCONN

The socket is connection-mode but isn't connected.

ENOTSOCK

The argument *s* isn't a socket.

EOPNOTSUPP

The *s* argument is associated with a socket that doesn't support one or more of the values set in *flags*.

EPIPE

The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type `SOCK_STREAM`, a `SIGPIPE` signal is generated to the calling thread.

EWOULDBLOCK

The socket is marked nonblocking and the requested operation would block.

If the address family of the socket is `AF_UNIX`, `sendmsg()` fails if:

EIO

An I/O error occurred while reading from or writing to the filesystem.

ELOOP

A loop exists in symbolic links encountered during resolution of the pathname in the socket address, or more than `SYMLOOP_MAX` symbolic links were encountered during the resolution of the pathname in the socket address.

ENAMETOOLONG

A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX` characters, or pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ENOENT

A component of the pathname doesn't name an existing file, or the path name is an empty string.

ENOTDIR

A component of the path prefix of the pathname in the socket address isn't a directory.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

sendto()

Send a message to a socket at a specific address

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto( int s,
                const void * msg,
                size_t len,
                int flags,
                const struct sockaddr * to,
                socklen_t tolen );
```

Arguments:

s

The descriptor for the socket; see [socket\(\)](#) (p. 3075).

msg

A pointer to the message that you want to send.

len

The length of the message.

flags

A combination of the following:

- **MSG_OOB** — process out-of-band data. Use this bit when you send “out-of-band” data on sockets that support this notion (e.g. **SOCK_STREAM**). The underlying protocol must also support out-of-band data.
- **MSG_DONTROUTE** — bypass routing; create a direct interface. You normally use this bit only in diagnostic or routing programs.
- **MSG_NOSIGNAL** — don't raise a **SIGPIPE** signal when the other end breaks the connection.

to

A pointer to a **sockaddr** object that specifies the address of the target.

tolen

A `socklen_t` object that specifies the size of the *to* address.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `sendto()` function is used to transmit a message to another socket. You can use [send\(\)](#) (p. 2829) only when the socket is in a *connected* state; you can use `sendto()` at any time.

The address of the target is given by *to*, with *toLen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error `EMSGSIZE` is returned, and the message isn't transmitted.

No indication of failure to deliver is implicit in a `sendto()`. Locally detected errors are indicated by a return value of `-1`.

If no message space is available at the socket to hold the message to be transmitted, then `sendto()` normally blocks, unless the socket has been placed in nonblocking I/O mode. You can use [select\(\)](#) (p. 2791) to determine when it's possible to send more data.

Returns:

The number of bytes sent, or `-1` if an error occurs ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied for a component of the path prefix, or write access to the named socket is denied.

EAFNOSUPPORT

Addresses in the specified address family can't be used with this socket.

EAGAIN

The socket's file descriptor is marked `O_NONBLOCK`, and the requested operation would block.

EBADF

An invalid descriptor was specified.

ECONNRESET

A connection was forcibly closed by a peer.

EDESTADDRREQ

The socket isn't connection-mode and doesn't have its peer address set, and no destination address was specified.

EFAULT

An invalid user space address was specified for a parameter.

EHOSTUNREACH

The destination host can't be reached (probably because the host is down or a remote router can't reach it).

EINTR

A signal interrupted *sendto()* before any data was transmitted.

EINVAL

The *toLen* argument isn't a valid length for the address family.

EIO

An I/O error occurred while reading from or writing to the filesystem.

EISCONN

A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.

EMSGSIZE

The socket requires that the message be sent atomically, but the size of the message made this impossible.

ENETDOWN

The local network interface used to reach the destination is down.

ENETUNREACH

No route to the network is present.

ENOBUFS

The system couldn't allocate an internal buffer. The operation may succeed when buffers become available.

ENOMEM

Insufficient memory was available to fulfill the request.

ENOTCONN

The socket is connection-mode but isn't connected.

ENOTSOCK

The argument *s* isn't a socket.

EOPNOTSUPP

The *s* argument is associated with a socket that doesn't support one or more of the values set in *flags*.

EPIPE

The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type `SOCK_STREAM`, a `SIGPIPE` signal is generated to the calling thread.

EWOULDBLOCK

The socket is marked nonblocking and the requested operation would block.

If the address family of the socket is `AF_UNIX`, *sendto()* fails if:

EIO

An I/O error occurred while reading from or writing to the filesystem.

ELOOP

A loop exists in symbolic links encountered during the resolution of the pathname in the socket address, or more than `SYMLINK_MAX` symbolic links were encountered during the resolution of the pathname in the socket address.

ENAMETOOLONG

A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX` characters, or pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ENOENT

A component of the pathname doesn't name an existing file, or the pathname is an empty string.

ENOTDIR

A component of the path prefix of the pathname in the socket address isn't a directory.

Classification:*POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

servent

Structure for information from the services database

Synopsis:

```
#include <netdb.h>

struct servent {
    char *   s_name;
    char **  s_aliases;
    int      s_port;
    char *   s_proto;
};
```

Description:

This structure is used to hold the broken-out fields of a line in the network services database, `/etc/services`. The members of this structure are:

s_name

The name of the service.

s_aliases

A zero-terminated list of alternate names for the service.

s_port

The port number that the service resides at. Port numbers are returned in network byte order.

s_proto

The name of the protocol to use when contacting the service.

Classification:

[*POSIX 1003.1*](#) (p. 46)

set_lowest_fd()

Specify the lowest acceptable file descriptor

Synopsis:

```
#include <fcntl.h>

int set_lowest_fd( int fd );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Arguments:

fd

The minimum file descriptor that you want to get.

Description:

The *set_lowest_fd()* function sets the lowest file descriptor number that will be returned by [open\(\)](#) (p. 1905), [pipe\(\)](#) (p. 2011), and other functions that return file descriptors. A process with the setuid/setgid file permission bit set can use this function at the beginning of *main()* to ensure that it never unexpectedly gets back a file descriptor equal to 0, 1, or 2, which malicious programmers sometimes cause to happen by closing *stdin*, *stdout*, or *stderr* before spawning the process.



This function doesn't affect [dup](#) (p. 483) and [dup2](#) (p. 485) because they're primarily used to set file descriptors 0, 1, and 2 for file-redirection purposes.

Returns:

The previous lowest number (0 for the first time it's called).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	No

setbuf()

Associate a buffer with a stream

Synopsis:

```
#include <stdio.h>

void setbuf( FILE *fp,
             char *buffer );
```

Arguments:

fp

The stream that you want to associate with a buffer.

buffer

NULL, or a pointer to the buffer; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setbuf()` function associates the supplied *buffer* with the stream specified by *fp*. If you want to call `setbuf()`, you must call it after opening the stream, but before doing any reading, writing, or seeking.

If *buffer* is NULL, all input/output for the stream is completely unbuffered. If *buffer* isn't NULL, then it must point to an array that's at least BUFSIZ characters long, and all input/output is fully buffered.



As a QNX Neutrino extension, you can use the **STDIO_DEFAULT_BUFSIZE** environment variable to override BUFSIZ as the default buffer size for stream I/O. The value of **STDIO_DEFAULT_BUFSIZE** must be greater than that of BUFSIZ.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main( void )
{
    char *buffer;
    FILE *fp;

    buffer = (char *)malloc( BUFSIZ );
    if( buffer == NULL ) {
        return EXIT_FAILURE;
    }

    fp = fopen( "some_file", "r" );
    setbuf( fp, buffer );

    /* . */
    /* . */
    /* . */

    fclose( fp );

    free( buffer );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

setbuffer()

Assign block buffering to a stream

Synopsis:

```
#include <unix.h>

void setbuffer( FILE *iop,
               char *abuf,
               size_t asize );
```

Arguments:

iop

The stream that you want to set the buffering for.

abuf

NULL, or a pointer to the buffer that you want the stream to use.

asize

The size of the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setbuffer()` and [setlinebuf\(\)](#) (p. 2878) functions assign buffering to a stream. The types of buffering available are:

Unbuffered

Information appears on the destination file or terminal as soon as written.

Block-buffered

Many characters are saved and written as a block.

Line-buffered

Characters are saved until either a newline is encountered or input is read from *stdin*.

You can use [fflush\(\)](#) (p. 641) to force the block out early. Normally all files are block-buffered. A buffer is obtained from [malloc\(\)](#) (p. 1569) when you perform the first [getc\(\)](#) (p. 823) or [putc\(\)](#) (p. 2447) on the file. If the standard stream *stdout* refers to a terminal, it's line-buffered. The standard stream *stderr* is unbuffered by default.

If you want to use [setbuffer\(\)](#), you must call it after opening the stream, but before doing any reading or writing. It uses the character array *abuf*, whose size is given by *asize*, instead of an automatically allocated buffer. If *abuf* is NULL, input and output are completely unbuffered. A manifest constant **BUFSIZ**, defined in the `<stdio.h>` header, tells how large an array is needed:

```
char buf[BUFSIZ];
```



As a QNX Neutrino extension, you can use the **STDIO_DEFAULT_BUFSIZE** environment variable to override **BUFSIZ** as the default buffer size for stream I/O. The value of **STDIO_DEFAULT_BUFSIZE** must be greater than that of **BUFSIZ**.

You can use [freopen\(\)](#) (p. 721). to change a stream from unbuffered or line-buffered to block buffered. To change a stream from block-buffered or line-buffered to unbuffered, call [freopen\(\)](#), and then call [setbuf\(\)](#) (p. 2845) with a buffer argument of NULL.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

setdomainname()

Set the domain name of the current host

Synopsis:

```
#include <unistd.h>

int setdomainname( const char * name,
                   size_t namelen );
```

Arguments:

name

The domain name.

namelen

The length of the name.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *setdomainname()* function sets the domain *name* of the host machine. Only the superuser (root) can use this function and even then, the function is normally used only when bootstrapping a system.

Returns:

0

Success.

-1

Failure; *errno* (p. 539) is set.

Errors:

EFAULT

The *name* or *namelen* parameters gave an invalid address.

EPERM

The caller tried to set the domain name without being the superuser.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setegid()

Set the effective group ID for a process

Synopsis:

```
#include <unistd.h>

int setegid( gid_t gid );
```

Arguments:

gid

The effective group ID that you want to use for the process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setegid()* function lets the calling process set the effective group ID based on the following:

- If the process has the `PROC_MGR_AID_SETGID` ability enabled (see [procmgr_ability\(\)](#) (p. 2141)), the *setegid()* function sets the effective group ID to *gid*.
- If the process doesn't have the `PROC_MGR_AID_SETGID` ability enabled, but *gid* is equal to the real group ID or saved set-group ID, *setegid()* sets the effective group ID to *gid*.

The real and saved group ID aren't changed.



If a set-group ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group ID.

Returns:

Zero for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of *gid* is out of range.

EPERM

The process doesn't have the `PROC_MGR_AID_SETGID` ability enabled, and *gid* doesn't match the real group ID or the saved set-group ID.

Examples:

```
/*
 * This process sets its effective group ID to 2
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    gid_t oegid;

    oegid = getegid();
    if( setegid( 2 ) == -1 ) {
        perror( "setegid" );
        return EXIT_FAILURE;
    }

    printf( "Was effective group %d, is 2\n", oegid );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setenv()

Create or change an environment variable

Synopsis:

```
#include <stdlib.h>

int setenv( const char* name,
            const char* value,
            int overwrite );
```

Arguments:

name

The name of the environment variable that you want to set.

value

NULL, or the value for the environment variable; see below.

overwrite

A nonzero value if you want the function to overwrite the variable if it exists, or 0 if you don't want to overwrite the variable.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `setenv()` function sets the environment variable *name* to *value*. If *name* doesn't exist in the environment, it's created; if *name* exists and *overwrite* is nonzero, the variable's old value is overwritten with *value*; otherwise, it isn't changed.



This function doesn't free any memory. If you want to change the value of an existing environment variable, you should use [putenv\(\)](#) (p. 2454) instead.

Copies of the specified *name* and *value* are placed in the environment.

If *value* is NULL, the environment variable specified by *name* is removed from the environment.



The value of the global *environ* pointer could be changed by a call to the *setenv()* function.

Environment variable names are case-sensitive.

Returns:

0

Success.

Nonzero

An error occurred (*errno* (p. 539) is set).

Errors:

ENOMEM

Not enough memory to allocate a new environment variable.

Examples:

Change the string assigned to **INCLUDE** and then display the new string:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char* path;

    if( setenv( "INCLUDE",
               "/usr/nto/include:/home/fred/include",
               1 ) == 0 ) {
        if( (path = getenv( "INCLUDE" )) != NULL ) {
            printf( "INCLUDE=%s\n", path );
        }
    }

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The *setenv()* function manipulates the environment pointed to by the global [environ](#) (p. 528) variable.

seteuid()

Set the effective user ID

Synopsis:

```
#include <unistd.h>

int seteuid( uid_t uid );
```

Arguments:

uid

The effective user ID that you want to use for the process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *seteuid()* function lets the calling process set the effective user ID, based on the following:

- If the process has the `PROC_MGR_AID_SETUID` ability enabled (see [procmgr_ability\(\)](#) (p. 2141)), the *seteuid()* function sets the effective user ID to *uid*.
- If the process doesn't have the `PROC_MGR_AID_SETUID` ability enabled, but *uid* is equal to the real user ID or saved set-user ID, *seteuid()* sets the effective user ID to *uid*.

The real and saved user IDs aren't changed.



If a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-UID.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of *uid* is out of range.

EPERM

The process doesn't have the PROC_MGR_AID_SETUID ability enabled, and *uid* doesn't match the real user ID or the saved set-user ID.

Examples:

```
/*
 * This process sets its effective userid to 0 (root).
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    uid_t oeuid;

    oeuid = geteuid();
    if( seteuid( 0 ) == -1 ) {
        perror( "seteuid" );
        return EXIT_FAILURE;
    }

    printf( "effective userid now 0, was %d\n",
        oeuid );

    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setgid()

Set the real, effective and saved group IDs

Synopsis:

```
#include <unistd.h>

int setgid( gid_t gid );
```

Arguments:

gid

The group ID that you want to use for the process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setgid()* function lets the calling process set the real, effective and saved group IDs, based on the following:

- If the process has the `PROC_MGR_AID_SETGID` ability enabled (see [procmgr_ability\(\)](#) (p. 2141)), the *setgid()* function sets the real group ID, effective group ID and saved group ID to *gid*.
- If the process doesn't have the `PROC_MGR_AID_SETGID` ability enabled, but *gid* is equal to the real group ID, *setgid()* sets the effective group ID to *gid*; the real and saved group IDs aren't changed.

This function doesn't change any supplementary group IDs of the calling process.

If you wish to change only the effective group ID, you should consider using the *setegid()* function.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set to indicate the error.

Errors:

EINVAL

The value of *gid* is invalid.

EPERM

The process doesn't have the PROC_MGR_AID_SETGID ability enabled, and *gid* doesn't match the real group ID.

Examples:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    gid_t ogid;

    ogid = getgid();
    if( setgid( 2 ) == -1 ) {
        perror( "setgid" );
        return EXIT_FAILURE;
    }
    printf( "group id is now 2, was %d\n", ogid );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setfsent()

Open the filesystem table (/etc/fstab) file

Synopsis:

```
#include <fstab.h>

int setfsent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *setfsent()* routine opens the filesystem table file, `/etc/fstab`, or rewinds the file if it's already open.

Returns:

1

Success.

0

An error occurred; [errno](#) (p. 539) is set.

Classification:

[NetBSD](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The functions that work with `/etc/fstab` use static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

setgrent()

Rewind to the start of the group database file

Synopsis:

```
#include <grp.h>

int setgrent( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *setgrent()* function rewinds to the start of the group name database file. It's provided for programs that make multiple lookups in the group database (using the [getgrgid\(\)](#) (p. 853) and [getgrnam\(\)](#) (p. 857) calls) to avoid the default opening and closing of the group database for each access.

Returns:

- 0
Success.
- 1
An error occurred.

Errors:

The *setgrent()* function uses *fopen()*. As a result, [errno](#) (p. 539) can be set to an error for the [fopen\(\)](#) (p. 681) call.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

setgroups()

Set supplementary group IDs

Synopsis:

```
#include <unistd.h>

int setgroups( int ngroups,
               const gid_t *gidset );
```

Arguments:

ngroups

The number of entries in the *gidset* array.

gidset

An array of the supplementary group IDs that you want to assign to the current user. This number of entries in this array can't exceed `sysconf(_SC_NGROUPS_MAX)`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setgroups()* function sets the group access list of the current user to the array of group IDs in *gidset*.



In order to set new groups or delete existing groups, your process must have the `PROCmgr_AID_SETGID` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

The *gidset* argument isn't a valid pointer.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

sethostent()

Open the host database file

Synopsis:

```
#include <netdb.h>

void sethostent( int stayopen );
```

Arguments:

stayopen

Nonzero if you want all queries to the name server to use TCP and you want the connection to be retained after each call to [gethostbyname\(\)](#) (p. 872) or [gethostbyaddr\(\)](#) (p. 866).

If the *stayopen* flag is zero, queries use UDP datagrams.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *sethostent()* routine opens the host database file.

You can use the *sethostent()* function to request the use of a connected TCP socket for queries.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

sethostname()

Set the name of the current host

Synopsis:

```
#include <unistd.h>

int sethostname( const char * name,
                 size_t namelen );
```

Arguments:

name

The name that you want to use for the host machine. Hostnames are limited to MAXHOSTNAMELEN characters (defined in <sys/param.h>).

namelen

The length of the name.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *sethostname()* function sets the name of the host machine to be *name*. Only the superuser can call this function; this is normally done only at boot time.



This function sets the value of the `_CS_HOSTNAME` configuration string, not that of the ***HOSTNAME*** environment variable.

Returns:

0

Success.

-1

An error occurred ([*errno*](#) (p. 539) is set).

Errors:**EFAULT**

Either *name* or *namelen* gave an invalid address.

EPERM

Although the caller wasn't the superuser, it tried to set the hostname.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is restricted to the superuser, and is normally used only at boot time.

SETIOV()

Fill in the fields of an `iov_t` structure

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

void SETIOV( iov_t *msg,
             void *addr,
             size_t len );
```

Arguments:

- msg**
A pointer to the `iov_t` structure structure that you want to set.
- addr**
The value you want to use for the structure's `iov_base` member.
- len**
The value you want to use for the structure's `iov_len` member.

Description:

The `SETIOV()` macro fills in the fields of an `iov_t` message structure. The `iov_t` structure consists of two fields:

```
typedef struct iovec {
    void      *iov_base;
    size_t    iov_len;
} iov_t;
```



`SETIOV()` doesn't make a copy of the data that `addr` points to; it just copies the pointer.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

setitimer()

Set the value of an interval timer

Synopsis:

```
#include <sys/time.h>

int setitimer ( int which,
                const struct itimerval *value,
                struct itimerval *ovalue );
```

Arguments:

which

The interval time whose value you want to set. Currently, this must be `ITIMER_REAL`.

value

A pointer to a `itimerval` structure that specifies the value that you want to set the interval timer to.

ovalue

`NULL`, or a pointer to a `itimerval` structure where the function can store the old value of the interval timer.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The system provides each process with interval timers, defined in `<sys/time.h>`. The `setitimer()` function sets the value of the timer specified by *which* to the value specified in the structure pointed to by *value*, and if *ovalue* isn't `NULL`, stores the previous value of the timer in the structure it points to.

A timer value is defined by the `itimerval` structure, which includes the following members:

```
struct timeval  it_interval;    /* timer interval */
struct timeval  it_value;       /* current value */
```

Each `struct timeval` contains the following members:

- `time_t tv_sec` — the number of seconds since the start of the Unix Epoch.
- `suseconds_t tv_usec` — the number of microseconds.

The `it_value` member indicates the time to the next timer expiration. The `it_interval` member specifies a value to be used in reloading `it_value` when the timer expires. Setting `it_value` to 0 disables a timer, regardless of the value of `it_interval`. Setting `it_interval` to 0 disables a timer after its next expiration (assuming `it_value` is nonzero).

Time values smaller than the resolution of the system clock are rounded up to the resolution of the system clock.

The only supported timer is `ITIMER_REAL`, which decrements in real time. A `SIGALRM` signal is delivered when this timer expires.

The `SIGALRM` so generated isn't maskable on this bound thread by any signal-masking function, `pthread_sigmask()`, or `sigprocmask()`.



Because of the nature of time measurement, the timer might actually expire later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EINVAL

The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the *which* argument is unrecognized.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

All flags to *setitimer()* other than `ITIMER_REAL` behave as documented only with “bound” threads. Their ability to mask the signal works only with bound threads. If the call is made using one of these flags from an unbound thread, the system call returns -1 and sets *errno* to `EACCES`.

These behaviors are the same for bound or unbound POSIX threads. A POSIX thread with system-wide scope, created by the call:

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

is equivalent to a Solaris bound thread. A POSIX thread with local process scope, created by the call:

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

is equivalent to a Solaris unbound thread.

The microseconds field shouldn't be equal to or greater than one second.

The *setitimer()* function is independent of [alarm\(\)](#) (p. 144).

Don't use *setitimer(ITIMER_REAL)* with the [sleep\(\)](#) (p. 3028) routine. A *sleep()* call wipes out knowledge of the user signal handler for `SIGALRM`.

The granularity of the resolution of the alarm time is platform-dependent.

setjmp()

Save the calling environment for longjmp()

Synopsis:

```
#include <setjmp.h>

int setjmp( jmp_buf env );
```

Arguments:

env

A buffer where the function can save the calling environment.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setjmp()` function saves the calling environment in its `env` argument for use by the [longjmp\(\)](#) (p. 1549) function.

Error handling can be implemented by using `setjmp()` to record the point to return to following an error. When an error is detected in a function, that function uses `longjmp()` to jump back to the recorded position. The original function that called `setjmp()` must still be active (that is, it can't have returned to the function that called it).

Be careful to ensure that any resources (allocated memory, opened files, etc) are cleaned up properly.



Do *not* use [longjmp\(\)](#) (p. 1549) or [siglongjmp\(\)](#) (p. 2970) to restore an environment saved by a call to `setjmp()` or [sigsetjmp\(\)](#) (p. 3010) in another thread. If you're lucky, your application will crash; if not, it'll look as if it works for a while, until random scribbling on the stack causes it to crash.

Returns:

Zero on the first call, or nonzero if the return is the result of a call to the `longjmp()` function.

Examples:

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf env;

void rtn( void )
{
    printf( "about to longjmp()\n" );
    longjmp( env, 14 );
}

int main( void )
{
    int ret_val;

    ret_val = setjmp( env );

    if( ret_val == 0 ) {
        printf( "after setjmp(): %d\n", ret_val );
        rtn();
        printf( "back from rtn(): %d\n", ret_val );
    } else {
        printf( "back from longjmp(): %d\n", ret_val );
    }

    return EXIT_SUCCESS;
}
```

produces the output:

```
after setjmp(): 0
about to longjmp()
back from longjmp(): 14
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setkey()

Set the key used in encryption

Synopsis:

```
#include <stdlib.h>

void setkey( const char *__key );
```

Arguments:

`__key`

A 64-character array of binary values (numeric 0 or 1).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `setkey()` function allows limited access to the NBS Data Encryption Standard (DES) algorithm itself. It derives a 56-bit key from the given `__key` by dividing the array into groups of 8 and ignoring the last bit in each group.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

setlinebuf()

Assign line buffering to a stream

Synopsis:

```
#include <unix.h>

int setlinebuf( FILE *iop );
```

Arguments:

iop

The stream that you want to use line buffering.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [setbuffer\(\)](#) (p. 2847) and *setlinebuf()* functions assign buffering to a stream. The types of buffering available are:

Unbuffered

Information appears on the destination file or terminal as soon as written.

Block-buffered

Many characters are saved and written as a block.

Line-buffered

Characters are saved until either a newline is encountered or input is read from *stdin*.

You can use [fflush\(\)](#) (p. 641) to force the block out early. Normally all files are block-buffered. A buffer is obtained from [malloc\(\)](#) (p. 1569) when the first [getc\(\)](#) (p. 823) or [putc\(\)](#) (p. 2447) is performed on the file. If the standard stream *stdout* refers to a terminal, it's line-buffered. The standard stream *stderr* is unbuffered by default.

You can use *setlinebuf()* to change the buffering on a stream from block-buffered or unbuffered to line-buffered. Unlike *setbuffer()*, you can call *setlinebuf()* at any time that the stream *iop* is active.

You can use [freopen\(\)](#) (p. 721). to change a stream from unbuffered or line-buffered to block buffered. To change a stream from block-buffered or line-buffered to unbuffered, call *freopen()*, and then call [setbuf\(\)](#) (p. 2845) with a buffer argument of `NULL`.

Returns:

No useful value.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

setlocale()

Select a program's locale

Synopsis:

```
#include <locale.h>

char * setlocale( int category,
                  const char * locale );
```

Arguments:

category

The part of the environment that you want to set; one of:

- LC_ALL — select the entire locale environment.
- LC_COLLATE — select only the collating sequence.
- LC_CTYPE — select only the character-handling information.
- LC_MESSAGES — specify the language to be used for messages.
- LC_MONETARY — select only monetary formatting information.
- LC_NUMERIC — select only the numeric-format environment.
- LC_TIME — select only the time-related environment.

locale

The locale that you want to use. The following built-in locales are offered:

- C (default)
- C-TRADITIONAL
- POSIX

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setlocale()* function selects a program's *locale*, according to the specified *category* and the specified *locale*.

A locale affects several things:

- The collating sequence (the order in which characters compare with one another) used by [strcoll\(\)](#) (p. 3190) or [wcscoll\(\)](#) (p. 3678).
- The way certain character-handling functions (such as [isalnum\(\)](#) (p. 1421) and [isalpha\(\)](#) (p. 1423)) operate. The wide-character versions include [iswalnum\(\)](#) (p. 1455) and [iswalpha\(\)](#) (p. 1457).
- The decimal-point character used in formatted input/output and string conversion ([printf\(\)](#) (p. 2133), [scanf\(\)](#) (p. 2713), and friends).
- The format and names used in the string produced by the [strftime\(\)](#) (p. 3202) and [wcsftime\(\)](#) (p. 3684) functions.

See the [localeconv\(\)](#) (p. 1526) function for more information about the locale.

At the start of a program, the default C locale is initialized as if the following call to [setlocale\(\)](#) appeared at the start of [main\(\)](#) (p. 1566):

```
(void)setlocale( LC_ALL, "C" );
```

Returns:

The string associated with the specified *category* for the new *locale*, or NULL if an error occurs. This function doesn't change the program's *locale* when error occurs.



For a NULL pointer of *locale*, this function returns a pointer to the string associated with the *category* for the program's current *locale*. This function doesn't change the program's *locale*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

setlogmask()

Set the system log priority mask

Synopsis:

```
#include <syslog.h>

int setlogmask( int maskpri );
```

Arguments:

maskpri

The new log priority mask; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setlogmask()* function sets the log priority mask to *maskpri* and returns the previous mask. Calls to [syslog\(\)](#) (p. 3316) or [vsyslog\(\)](#) (p. 3643) with a priority that isn't set in *maskpri* are rejected.

You can calculate the mask for an individual priority *pri* with the macro:

```
LOG_MASK(pri);
```

You can get the mask for all priorities up to and including *toppri* with the macro:

```
LOG_UPTO(toppri);
```

The default allows all priorities to be logged. See the *syslog()* function for a list of the priorities.

Returns:

The previous log mask level.

Examples:

See [syslog\(\)](#) (p. 3316).

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

setnetent()

Open the network name database file

Synopsis:

```
#include <netdb.h>

void setnetent( int stayopen );
```

Arguments:

stayopen

Nonzero if you don't want the network database to be closed after each call to [getnetbyname\(\)](#) (p. 900) or [getnetbyaddr\(\)](#) (p. 898).

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *setnetent()* function opens and rewinds the network name database file.

Files:

/etc/networks

Network name database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, you should copy it before any subsequent calls overwrite it.

setpgid()

Join or create a process group

Synopsis:

```
#include <process.h>

int setpgid( pid_t pid,
             pid_t pgid );
```

Arguments:

pid

0, or the ID of the process whose process group you want to set.

pgid

0, or the process group ID that you want to join or create.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setpgid()* function is used either to join an existing process group or to create a new process group within the session of the calling process. The process group ID of a session leader doesn't change.

The following definitions are worth mentioning:

Process

An executing instance of a program, identified by a nonnegative integer called a process ID.

Process group

A collection of one or more processes, with a unique process group ID. A process group ID is a positive integer.

On successful completion, the process group ID of the process with a process ID matching *pid* is set to *pgid*. As a special case, you can specify either *pid* or *pgid* as zero, meaning that the process ID of the calling process is to be used.

Returns:

0

Success.

-1

An error occurred; *errno* (p. 539) is set.**Errors:****EACCES**

The value of the *pid* argument matches the process ID of a child process of the calling process, and the child process has successfully executed one of the *exec*()* functions.

EINVAL

The value of *pgid* is less than zero.

ENOSYS

The *setpgid()* function isn't supported by this implementation (included for POSIX compatibility).

EPERM

The calling process doesn't have sufficient privilege to set the process group id *pgid* on process *pid*.

ESRCH

The process *pid* doesn't exist.

Examples:

```
/*
 * The process joins process group 0.
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <process.h>

int main( void )
{
    if( setpgid( getpid(), 0 ) == -1 ) {
        perror( "setpgid" );
    }
    printf( "%d belongs to process group %d\n",
            getpid(), getpgrp() );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setpgroup()

Set the process group

Synopsis:

```
#include <unistd.h>

pid_t setpgroup( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

If the calling process isn't already a session leader, *setpgroup()* makes it one by setting its process group ID and session ID to the value of its process ID, and releases its controlling terminal.



In order to successfully call this function, your process must have the PROC_MGR_AID_PGRP ability enabled. This ability is enabled by default (for POSIX conformance). You can disable it completely or restrict it to specific pid ranges. For more information, see [proc_mngr_ability\(\)](#) (p. 2141).

Returns:

The new process group ID.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setprio()

Set the priority of a process

Synopsis:

```
#include <sched.h>

int setprio( pid_t pid,
             int prio );
```

Arguments:

pid

The process ID of the process whose priority you want to set.

prio

The new priority.

In QNX Neutrino 6.6 or later, you can wrap the new priority in one these macros to specify how to handle out-of-range priority requests:

- *SCHED_PRIO_LIMIT_ERROR(prio)* — indicate an error
- *SCHED_PRIO_LIMIT_SATURATE(prio)* — saturate at the maximum allowed priority

If *procnto* was started with a *-P* option ending with *s* or *S*, out-of-range priority requests by default saturate at the maximum allowed value.

Library:

libc

Use the *-l c* option to *qcc* to link against this library. This library is usually included automatically.

Description:

The *setprio()* function changes the priority of thread 1 of process *pid* to priority *prio*. If *pid* is zero, the priority of the calling thread is set.



In order to change its priority to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The previous priority, or -1 if an error occurred (*errno* is set).

Errors:

EINVAL

The priority *prio* isn't a valid priority.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process *pid* doesn't exist.

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The *getprio()* and *setprio()* functions are included in the libraries for porting QNX 4 applications. For new programs, use [sched_setparam\(\)](#) (p. 2738), [pthread_setschedparam\(\)](#) (p. 2400), or [pthread_setschedprio\(\)](#) (p. 2402).

setprotoent()

Open the protocol name database file

Synopsis:

```
#include <netdb.h>

void setprotoent( int stayopen );
```

Arguments:

stayopen

Nonzero if you don't want the database to be closed after each call to [getprotobyname\(\)](#) (p. 924) or [getprotobynumber\(\)](#) (p. 926).

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *setprotoent()* function opens and rewinds the protocol name database file. If the *stayopen* flag is nonzero, the protocol name database isn't closed after each call to [getprotobyname\(\)](#) (p. 924) or [getprotobynumber\(\)](#) (p. 926).

Files:

/etc/protocols

Protocol name database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, you should copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

setpwent()

Rewind the password database file

Synopsis:

```
#include <sys/types.h>
#include <pwd.h>

int setpwent( void );
```

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `setpwent()` function rewinds to the start of the password name database file. It's provided for programs that make multiple lookups in the password database (using the [getpwuid\(\)](#) (p. 938) and [getpwnam\(\)](#) (p. 934) calls) to avoid opening and closing the password database for each access.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

setregid()

Set real and effective group IDs

Synopsis:

```
#include <unistd.h>

int setregid( gid_t rgid,
              gid_t egid );
```

Arguments:

rgid

The real group ID that you want to use for the process, or -1 if you don't want to change it.

egid

The effective group ID that you want to use for the process, or -1 if you don't want to change it.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *setregid()* function sets the real and effective group IDs of the calling process. If *rgid* or *egid* is -1, the corresponding real or effective group ID is left unchanged.

If the process has the PROC_MGR_AID_SETGID ability enabled (see [procmgr_ability\(\)](#) (p. 2141)), you can set the real group ID and the effective group ID to any legal value.

If the process doesn't have the PROC_MGR_AID_SETGID ability enabled, you can set either the real group ID to the saved set-group ID, or the effective group ID to either the saved set-group ID or the real group ID.



If a set-group ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group ID.

In either case, if you're changing the real group ID (i.e. *rgid* isn't -1), or you're changing the effective group ID to a value that isn't equal to the real group ID, the saved set-group ID is set to the new effective group ID.

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EINVAL

The *rgid* or *egid* is out of range.

EPERM

The calling process doesn't have the PROC_MGR_AID_SETGID ability enabled, and you tried to change the effective group ID to a value other than the real or saved set-group ID.

Or:

The calling process doesn't have the PROC_MGR_AID_SETGID ability enabled, and you tried to change the real group ID to a value other than the effective group ID.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setreuid()

Set the real and effective user IDs for the calling process

Synopsis:

```
#include <unistd.h>

int setreuid( uid_t ruid,
              uid_t euid );
```

Arguments:

ruid

The real user ID that you want to use for the process, or -1 if you don't want to change it.

euid

The effective user ID that you want to use for the process, or -1 if you don't want to change it.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setreuid()` function sets the real and effective user IDs of the calling process. If *ruid* or *euid* is -1, the corresponding real or effective user ID isn't changed.

If the process has the `PROC_MGR_AID_SETUID` ability enabled (see [procmgr_ability\(\)](#) (p. 2141)), you can set the real user ID and the effective user ID to any legal value.

If the process doesn't have the `PROC_MGR_AID_SETUID` ability enabled, you can set either the real user ID to the effective user ID, or the effective user ID to the saved set-user ID or the real user ID.



If a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-UID.

In either case, if you're changing the real user ID (i.e. *ruid* is not -1), or you're changing the effective user ID to a value that isn't equal to the real user ID, the saved set-user ID is set equal to the new effective user ID.

Returns:

Zero on success, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EINVAL

The *ruid* or *euid* is out of range.

EPERM

The calling process doesn't have the `PROC_MGR_AID_SETUID` ability enabled, and you tried to change the effective user ID to a value other than the real or saved set-user ID.

Or:

The calling process doesn't have the `PROC_MGR_AID_SETUID` ability enabled, and you tried to change the real user ID to a value other than the effective user ID.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setrlimit(), setrlimit64()

Set the limit on a system resource

Synopsis:

```
#include <sys/resource.h>

int setrlimit( int resource,
               const struct rlimit * rlp );

int setrlimit64( int resource,
                 const struct rlimit64 * rlp );
```

Arguments:

resource

The resource whose limit you want to set. For a list of the possible resources, their descriptions, and the actions taken when the current limit is exceeded, see below.

rlp

A pointer to a `rlimit` or `rlimit64` structure that specifies the limit that you want to set for the resource. The `rlimit` and `rlimit64` structures include at least the following members:

```
rlim_t rlim_cur; /* current (soft) limit */
rlim_t rlim_max; /* hard limit */
```

The `rlim_t` type is an arithmetic data type to which you can cast objects of type `int`, `size_t`, and `off_t` without loss of information.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `setrlimit()` and `setrlimit64()` functions set the limits on the consumption of a variety of system resources by a process and each process it creates. The `setrlimit64()` function is a 64-bit version of `setrlimit()`.

Each call to *setrlimit()* identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit.

A process can change soft limits to any value that's less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that's greater than or equal to the soft limit. Only a process with the `PROCGR_AID_RLIMIT` ability enabled (see *procgr_ability()* (p. 2141)) can raise a hard limit. Both hard and soft limits can be changed in a single call to *setrlimit()*, subject to the constraints described above. Limits may have an “infinite” value of `RLIM_INFINITY`.



`RLIM_INFINITY` is a special value, and its actual numerical value doesn't represent a valid size of virtual memory or the address space.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized below:

Resource	Description	Action
<code>RLIMIT_AS</code>	(POSIX) The maximum size, in bytes, of a process's mapped address space.	If the limit is exceeded, the <i>brk()</i> (p. 213), <i>mmap()</i> (p. 1656), and <i>sbrk()</i> (p. 2702) functions fail with <i>errno</i> (p. 539) set to <code>ENOMEM</code> . In addition, the automatic stack growth fails with the effects outlined above.
<code>RLIMIT_CORE</code>	(POSIX) The maximum size, in bytes, of a core file that a process may create. A limit of 0 prevents the creation of a core file.	The writing of a core file terminates at this size.
<code>RLIMIT_CPU</code>	(POSIX) The maximum amount of CPU time, in seconds, that a process may use. This is a soft limit only.	<code>SIGXCPU</code> is sent to the process. If the process is holding or ignoring <code>SIGXCPU</code> , the behavior is defined by the scheduling class.
<code>RLIMIT_DATA</code>	(POSIX) The maximum size, in bytes, that a process's heap may be. In QNX Neutrino, <code>RLIMIT_DATA</code> covers all	If the limit is exceeded, the <i>brk()</i> (p. 213), <i>mmap()</i> (p. 1656), and <i>sbrk()</i> (p. 2702) functions fail with <i>errno</i> (p. 539) set to <code>ENOMEM</code> .

Resource	Description	Action
	mappings made by the process that are MAP_ANON MAP_PRIVATE that aren't MAP_STACK, which corresponds typically to heap allocations.	
RLIMIT_FREEMEM	The limit on total memory usage, in tenths of a per cent of the total system memory. For example, a limit of 500 on a system with 1000 MB of memory would be 500 MB.	This limit is shared with all other processes; a memory allocation by this process fails if it would cause the total system memory usage of all processes to exceed this process's limit. The intention is to allow the memory pool to be crudely partitioned. For example, important system processes could be given an RLIMIT_FREEMEM value of RLIM_INFINITY, and all others a value of 800. That would partition memory into an 80% chunk that's shared by everybody and a 20% chunk that's shared by important system processes.
RLIMIT_MEMLOCK	The maximum amount of memory that can be locked into physical memory (so it will never be paged out).	
RLIMIT_NOFILE	(POSIX) One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors that a process may create.	

Resource	Description	Action
RLIMIT_NPROC	The maximum number of processes.	Attempts to create new processes fail.
RLIMIT_NTHR	The maximum number of threads.	Attempts to create threads (e.g. by calling pthread_create() (p. 2284) fail.
RLIMIT_OFIL	Same as RLIMIT_NOFILE.	
RLIMIT_RSS	The maximum resident set size for a process; the same as RLIMIT_AS.	Same as RLIMIT_AS.
RLIMIT_STACK	(POSIX) The maximum size, in bytes, that a process's stack may be. The system will not automatically grow the stack beyond this limit. Within a process, <i>setrlimit()</i> increases the limit on the size of your stack, but doesn't move current memory segments to allow for that growth. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used. Within a multithreaded process, <i>setrlimit()</i> has no impact on the stack size limit for the calling thread if the calling thread isn't the main thread. A call to <i>setrlimit()</i> for RLIMIT_STACK impacts only the main thread's stack, and should be made	The SIGSEGV signal is sent to the process. If the process is holding or ignoring SIGSEGV, or is catching SIGSEGV and hasn't made arrangements to use an alternate stack, the disposition of SIGSEGV is set to SIG_DFL before it's sent.

Resource	Description	Action
	only from the main thread, if at all.	
RLIMIT_VMEM	Same as RLIMIT_AS.	Same as RLIMIT_AS.

Because limit information is stored in the per-process information, the shell builtin `ulimit` command (see the entry for `ksh` in the *Utilities Reference*) must directly execute this system call if it's to affect all future processes created by the shell.

The values of the current limit of the following resources affect these parameters:

Resource	Parameter
RLIMIT_NOFILE	OPEN_MAX

When you're using the `setrlimit()` function, if the requested new limit is `RLIM_INFINITY`, there's no new limit; otherwise, if the requested new limit is `RLIM_SAVED_MAX`, the new limit is the corresponding saved hard limit; otherwise, if the requested new limit is `RLIM_SAVED_CUR`, the new limit is the corresponding saved soft limit; otherwise, the new limit is the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type `rlim_t`, then it's overwritten with the new limit.

The result of setting a limit to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR` is unspecified unless a previous call to [getrlimit\(\)](#) (p. 942) returned that value as the soft or hard limit for the corresponding resource limit.

A limit whose value is greater than `RLIM_INFINITY` is permitted.

The `exec*` family of functions also cause resource limits to be saved.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

The `rlp` argument points to an illegal address.

EINVAL

An invalid resource was specified, the new *rlim_cur* exceeds the new *rlim_max*, or the limit specified can't be lowered because current usage is already higher than the limit.

EPERM

The limit specified to *setrlimit()* would have raised the maximum limit value, and the calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

setrlimit() is [POSIX 1003.1 XSI](#) (p. 46); *setrlimit64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setservent()*Open the network services database file***Synopsis:**

```
#include <netdb.h>

void setservent( int stayopen );
```

Arguments:***stayopen***

Nonzero if you don't want the database to be closed after each call to [getservbyname\(\)](#) (p. 952) or [getservbyport\(\)](#) (p. 954).

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *setservent()* function opens and rewinds the network services database file. If the *stayopen* flag is nonzero, the network services database won't be closed after each call to [getservbyname\(\)](#) (p. 952) or [getservbyport\(\)](#) (p. 954).

Files:***/etc/services***

Network services database file.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

This function uses static data; if you need the data for future use, you should copy it before any subsequent calls overwrite it.

setsid()

Create a new session

Synopsis:

```
#include <unistd.h>

pid_t setsid( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setsid()* function creates a new session with the calling process becoming the process group leader with no controlling terminal. The process group ID of the calling process is set to the process ID of the calling process. The calling process is the only process in the new process group, and is also the only process in the new session.

If the calling process is already a process group leader, a new session isn't created and an error is returned.



In order to create a new session, your process must have the `PROC_MGR_AID_PGRP` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The new process group ID for the calling process, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EPERM

The calling process is already a process group leader, the process group ID of a process other than the calling process matches the process ID of the calling process, or the `PROC_MGR_AID_PGRP` ability is disabled.

Examples:

```
/*
 * You can only become a session leader if you are not
 * a process group leader that is the default for a
 * command run from the shell.
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    if( fork() )
    {
        if( setsid() == -1 )
            perror( "parent: setsid" );
        else
            printf( "parent: I am a session leader\n" );
    }
    else
    {
        if( setsid() == -1 )
            perror( "child: setsid" );
        else
            printf( "child: I am a session leader\n" );
    }
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setsockopt()

Set options associated with a socket

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt( int s,
               int level,
               int optname,
               const void * optval,
               socklen_t optlen );
```

Arguments:

s

The file descriptor of the socket that the option is to be applied on, as returned by [socket\(\)](#) (p. 3075).

level

The protocol layer that the option is to be applied to. In most cases, it's a socket-level option and is indicated by SOL_SOCKET.

optname

The option for the socket file descriptor.

optval

A pointer to the value of the option (in most cases, whether the option is to be turned on or off). If no option value is to be returned, *optval* may be NULL.

Most socket-level options use an `int` parameter for *optval*. Others, such as the [SO_LINGER](#) (p. 965), [SO_SNDTIMEO](#) (p. 967), and [SO_RCVTIMEO](#) (p. 966) options, use structures that also let you get data associated with the option.

optlen

A pointer to the length of the value of the option. This argument is a value-result parameter; you should initialize it to indicate the size of the buffer pointed to by *optval*.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *setsockopt()* function sets the options associated with a socket.

See [getsockopt\(\)](#) (p. 962) for more detailed information.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid file descriptor *s*.

EDOM

Value was set out of range.

EFAULT

The address pointed to by *optval* isn't in a valid part of the process address space.

EINVAL

No *optval* value was specified.

ENOPROTOOPT

The option is unknown at the level indicated.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

setspent()

Rewind the shadow password database file

Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

void setspent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setspent()* function rewinds to the start of the shadow password database file. It's provided for programs that make multiple lookups in the database (using the [getsprnam\(\)](#) (p. 974) call) to avoid opening and closing the shadow password database for each access.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

setstate()

Reset the state of a pseudo-random number generator

Synopsis:

```
#include <stdlib.h>

char *setstate( const char *state );
```

Arguments:

state

A pointer to the state array that you want to use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

Once the state of the pseudo-random number generator has been initialized, `setstate()` allows switching between state arrays. The array defined by the `state` argument is used for further random-number generation until `initstate()` is called or `setstate()` is called again. The `setstate()` function returns a pointer to the previous state array.

This function is used in conjunction with the following:

[*initstate\(\)*](#) (p. 1133)

Initialize the state of the pseudo-random number generator.

[*random\(\)*](#) (p. 2487)

Generate a pseudo-random number using a default state.

[*srandom\(\)*](#) (p. 3158)

Set the seed used by the pseudo-random number generator.

After initialization, you can restart a state array at a different point in one of two ways:

- Call *initstate()* with the desired seed, state array, and size of the array.
- Call *setstate()* with the desired state, then call *srandom()* with the desired seed.
The advantage of using both of these functions is that the size of the state array doesn't have to be saved once it's initialized.

Returns:

A pointer to the previous state array, or NULL if an error occurred.

Examples:

See [initstate\(\)](#) (p. 1133).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

settimeofday()

Set the time and date

Synopsis:

```
#include <sys/time.h>

int settimeofday( const struct timeval *when,
                  void *not_used );
```

Arguments:

when

A pointer to a `timeval` structure that specifies the the time that you want to set. The `struct timeval` contains the following members:

- `time_t tv_sec` — the number of seconds since the start of the Unix Epoch.
- `suseconds_t tv_usec` — the number of microseconds.

not_used

This pointer must be `NULL` or the behavior of `settimeofday()` is unspecified. This argument is provided only for backwards compatibility.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

This function sets the time and date to the values stored in the structure pointed to by *when*.



In order to set the clock, your process must have the `PROCmgr_AID_CLOCKSET` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

0, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EFAULT**

An error occurred while accessing the *when* buffer.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[Legacy Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The *settimeofday()* function is provided for compatibility with existing Unix code. You shouldn't use it in new code — use [clock_settime\(\)](#) (p. 319) instead.

setuid()

Set the real, effective and saved user IDs

Synopsis:

```
#include <unistd.h>

int setuid( uid_t uid );
```

Arguments:

uid

The user ID that you want to use for the process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *setuid()* function lets the calling process set the real, effective and saved user IDs based on the following:

- If the process has the `PROC_MGR_AID_SETUID` ability enabled (see [procmgr_ability\(\)](#) (p. 2141)), *setuid()* sets the real user ID, effective user ID and saved user ID to *uid*.
- If the process doesn't have the `PROC_MGR_AID_SETUID` ability enabled, but *uid* is equal to the real user ID or saved set-user ID, *setuid()* sets the effective user ID to *uid*; the real and saved user IDs aren't changed.



If a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-UID.

If you wish to change only the effective user ID, you should consider using the *seteuid()* function.

Returns:

0 for success, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EINVAL

The value of *uid* is out of range.

EPERM

The process doesn't have the PROC_MGR_AID_SETUID ability enabled, and *uid* doesn't match the real user ID or saved set-user ID.

Examples:

```
/*
 * This process sets its userid to 0 (root)
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    uid_t ouid;

    ouid = getuid();
    if( setuid( 0 ) == -1 ) {
        perror( "setuid" );
        return EXIT_FAILURE;
    }

    printf( "userid %d switched to 0\n", ouid );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setutent()

Return to the beginning of the user-information file

Synopsis:

```
#include <utmp.h>

void setutent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `setutent()` function resets the input stream to the beginning of the file specified in `_PATH_UTMP`. Do this before each search for a new entry if you want the entire file to be examined.

Files:

`_PATH_UTMP`

Specifies the user information file.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

setvbuf()

Associate a buffer with a stream

Synopsis:

```
#include <stdio.h>

int setvbuf( FILE *fp,
             char *buf,
             int mode,
             size_t size );
```

Arguments:

fp

The stream that you want to associate with a buffer.

buffer

NULL, or a pointer to the buffer; see below.

mode

How you want the stream to be buffered:

- `_IOFBF` — input and output are fully buffered.
- `_IOLBF` — output is line buffered (i.e. the buffer is flushed when a newline character is written, when the buffer is full, or when input is requested).
- `_IONBF` — input and output are completely unbuffered.

size

The size of the buffer.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `setvbuf()` function associates a buffer with the stream designated by `fp`. If you want to call `setvbuf()`, you must call it after opening the stream, but before doing any reading, writing, or seeking.

If `buf` isn't NULL, the buffer it points to is used instead of an automatically allocated buffer.



As a QNX Neutrino extension, you can use the **`STDIO_DEFAULT_BUFSIZE`** environment variable to override BUFSIZ as the default buffer size for stream I/O. The value of **`STDIO_DEFAULT_BUFSIZE`** must be greater than that of BUFSIZ.

Returns:

0

Success.

EINVAL

The `mode` argument isn't valid.

ENOMEM

The `buf` argument is NULL, `size` isn't 0, and there isn't enough memory available to allocate a buffer.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *buf;
    FILE *fp;

    fp = fopen( "file", "r" );
    buf = malloc( 1024 );
    setvbuf( fp, buf, _IOFBF, 1024 );

    /* work with fp */
    ...

    fclose( fp );

    /* This is OUR buffer, so we have
     * to free it. Do that AFTER
     * you've closed the file.
     */

    free( buf );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sfree()

Deallocate a block of memory

Synopsis:

```
#include <malloc.h>

void _sfree( void *ptr,
             size_t size );
```

Arguments:

ptr

NULL, or a pointer to the block of memory that you want to free.

size

The number of bytes to deallocate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

When the value of the argument *ptr* is NULL, the `_sfree()` function does nothing; otherwise, the `_sfree()` function deallocates *size* bytes of memory located by the argument *ptr*, which was previously returned by a call to the appropriate version of `_scalloc()` (p. 2709) or `_smalloc()` (p. 3068). After the call, the freed block is available for allocation.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Calling `_sfree()` on a pointer already deallocated by a call to `_sfree()` could corrupt the memory allocator's data structures.

The *size* *must* match the size of the allocated block.

shm_ctl()

Give special attributes to a shared memory object

Synopsis:

```
#include <sys/mman.h>

int shm_ctl( int fd,
             int flags,
             uint64_t paddr,
             uint64_t size );
```

Arguments:

fd

The file descriptor that's associated with the shared memory object, as returned by [shm_open\(\)](#) (p. 2935).

flags

One or more of the following bits, defined in `<sys/mman.h>`:

- SHMCTL_ANON — allocate anonymous memory.
- SHMCTL_GLOBAL — a hint that any mapping to the object could be global across all processes.



In order to use the SHMCTL_GLOBAL flag, your process must have the PROCmgr_AID_MEM_GLOBAL ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

- SHMCTL_HIGHUSAGE — a hint that the object is used a lot, and so the system should try to do things to speed up access to it.
- SHMCTL_ISADMA — memory should be suitable for ISA DMA (e.g. it should be below 16 MB, and shouldn't cross 64 KB boundaries).
- SHMCTL_LAZY — delay allocating the memory until it's referenced.

If you create anonymous shared memory objects (by calling [mmap\(\)](#) (p. 1656) with MAP_ANON | MAP_SHARED and a file descriptor of -1), a MAP_LAZY flag implicitly sets the SHMCTL_LAZY flag on the object.

- SHMCTL_LAZYWRITE — a hint that a mapping of this object could use lazy-writing mechanisms.
- SHMCTL_LOWERPROT — a hint that the system may map this object in such a way that it trades lower memory protection for better performance.

- **SHMCTL_NODEFRAG** — used with **SHMCTL_PHYS** | **SHMCTL_ANON** to specify that you want physically contiguous memory, but not so much that you're willing to wait for defragmentation. If contiguous memory isn't immediately available, the call gives an error of **ENOMEM**, and you can retry with just **SHMCTL_ANON** if you're willing to accept discontiguity.
- **SHMCTL_NOX64K** — memory shouldn't cross 64 KB boundaries.
- **SHMCTL_PHYS** — use physical address, or allocate physically contiguous memory if used with **SHMCTL_ANON**.



In order to use the **SHMCTL_PHYS** flag, your process must have the **PROCmgr_AID_MEM_PHYS** ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

- **SHMCTL_PRIV** — a hint that a mapping of this object may require privileged access.
- **SHMCTL_REPEAT** — create a strided physical object. For example:

```
shm_ctl( fd, SHMCTL_PHYS, baseaddr, vstride );  
shm_ctl( fd, SHMCTL_REPEAT, pstride, count );
```



Some of the bits have specific meanings for different processors. For more information, see “[Flags for ARM processors](#) (p. 2927)” and “[Flags for x86 processors](#) (p. 2927),” below.

paddr

A physical address to assign to the object, if you set **SHMCTL_PHYS** in *flags*.

size

The new size of the object, in bytes, regardless of the **ANON/PHYS** flag.

Library:

libc

Use the **-l c** option to **gcc** to link against this library. This library is usually included automatically.

Description:

The *shm_ctl()* function modifies the attributes of the shared memory object identified by the handle, *fd*. This handle is the value returned by [shm_open\(\)](#) (p. 2935).

- The combination SHMCTL_ANON | SHMCTL_PHYS has the same behavior as for [mmap\(\)](#) (p. 1656): it indicates that you want physically contiguous RAM to be allocated for the object.



- On ARM targets, once you've called *shm_ctl()* for a shared memory object, you can't resize it. You must unmap and unlink the shared object, and then recreate it.
- If you specify SHMCTL_PHYS in the flags, then *paddr* and *size* must be even multiples of the page size (`sysconf(_SC_PAGE_SIZE)`).

The [shm_ctl_special\(\)](#) (p. 2931) function is similar to *shm_ctl()*, but has an additional, processor-specific argument.

Flags for ARM processors

For ARMv7 processors:

- Objects created with *shm_ctl()* are always mapped into each process's address space. These mappings are inherited across calls to [fork\(\)](#) (p. 685).
- SHMCTL_GLOBAL is ignored, since all mappings are placed in the per-process address space.
- SHMCTL_PRIV and SHMCTL_LOWERPROT are ignored. All mappings are created in the per-process address space with user mode access protections.

Flags for x86 processors

For x86 processors:

- SHMCTL_LAZYWRITE causes Write Combining to be used. Write Combining is normally used for write-only purposes, such as a video buffer, as a high-performance alternative to PROT_NOCACHE:
 - Writes are significantly faster than to PROT_NOCACHE memory.
 - Reads have the same performance as from PROT_NOCACHE memory, so they could be slow.
 - If memory has been previously mapped as a cached memory type, the caches must be flushed, or undefined behavior may result.
 - Write Combining works on SMP systems.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The shared memory object is already “special.”

EINVAL

An invalid combination of flags was specified.

ENOMEM

One of the following:

- There wasn't enough memory.
- You specified SHMCTL_PHYS | SHMCTL_ANON | SHMCTL_NODEFRAG, and contiguous memory wasn't immediately available. You can retry with just SHMCTL_ANON if you're willing to accept discontiguity.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

The following examples go together. Run `sharephyscreator`, followed by `sharephysuser`.

The `sharephyscreator` process maps in an area of physical memory and then overlays it with a shared memory object. The `sharephysuser` process opens that shared memory object in order to access the physical memory.

```
/*
 * sharephyscreator.c
 *
 * This maps in an area of physical memory and then
 * overlays it with a shared memory object. This way, another process
 * can open that shared memory object in order to access the physical
 * memory. The other process in this case is sharephysuser.
 *
 * Note that the size and address that you pass to shm_ctl() must be
 * even multiples of the page size (sysconf(_SC_PAGE_SIZE)).
 *
 * For VGA color text mode video memory:
 * sharephyscreator /wally b8000
 * Note that for VGA color text mode video memory, each character
 * is followed by an attribute byte. Here we just use a space.
 */

#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/stat.h>

char      *progname = "sharephyscreator";

main( int argc, char *argv[] )
{
```

```

char      *text = "H e l l o   w o r l d ! ";
int       fd, memsize;
char      *ptr, *name;
uint64_t  physaddr;

if ( argc != 3 ) {
    printf( "Use: sharephyscreator shared_memory_object_name \
physical_address_in_hex\n" );
    printf( "Example: sharephyscreator wally b8000\n" );
    exit( EXIT_FAILURE );
}
name = argv[1];
physaddr = atoh(argv[2]);
memsize = sysconf( _SC_PAGE_SIZE ); /* this should be enough
                                     for our string */

/* map in the physical memory */

ptr = mmap_device_memory( 0, memsize, PROT_READ|PROT_WRITE, 0, physaddr );
if ( ptr == MAP_FAILED ) {
    printf( "%s: mmap_device_memory for physical address %llx failed: %s\n",
            progname, physaddr, strerror(errno) );
    exit( EXIT_FAILURE );
}

/* open the shared memory object, create it if it doesn't exist */

fd = shm_open( name, O_RDWR | O_CREAT, 0 );
if ( fd == -1 ) {
    printf( "%s: error creating the shared memory object '%s': %s\n",
            progname, name, strerror(errno) );
    exit( EXIT_FAILURE );
}

/* overlay the shared memory object onto the physical memory */

if ( shm_ctl( fd, SHMCTL_PHYS, physaddr, memsize ) == -1 ) {
    printf( "%s: shm_ctl failed: %s\n", progname, strerror(errno) );
    close( fd );
    munmap( ptr, memsize );
    shm_unlink( name );
    exit( EXIT_FAILURE );
}
strcpy( ptr, text ); /* write to the shared memory */

printf( "\n%s: Physical memory mapped in, shared memory overlaid onto it.\n"
        "%s: Wrote '%s' to physical memory.\n"
        "%s: Sleeping for 20 seconds. While this program is sleeping\n"
        "%s: run 'sharephysuser %s %d'.\n",
        progname, progname, ptr, progname, progname, name,
        strlen(text)+1 );
sleep( 20 );

printf( "%s: Woke up. Cleaning up and exiting ...\n", progname );

close( fd );
munmap( ptr, memsize );
shm_unlink( name );
}

```

The following is meant to be run with sharephyscreator.

```

/*
 * sharephysuser.c
 *
 * This one is meant to be run in tandem with sharephyscreator.
 *
 * Run it as: sharephysuser shared_memory_object_name length
 * Example: sharephysuser wally 49
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/stat.h>

char      *progname = "sharephysuser";

main( int argc, char *argv[] )
{
    int     fd, len, i;
    char    *ptr, *name;

    if ( argc != 3 ) {
        fprintf( stderr, "Use: sharephysuser shared_memory_object_name \
length\n" );
        fprintf( stderr, "Example: sharephysuser wally 49\n" );
        exit( EXIT_FAILURE );
    }
    name = argv[1];
    len = atoi( argv[2] );

    /* open the shared memory object */

    fd = shm_open( name, O_RDWR, 0 );
    if ( fd == -1 ) {
        fprintf( stderr, "%s: error opening the shared memory object '%s': %s\n",
                progname, name, strerror(errno) );
    }
}

```

```
    exit( EXIT_FAILURE );
}

/* get a pointer to a piece of the shared memory, note that we
   only map in the amount we need to */

ptr = mmap( 0, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0 );
if ( ptr == MAP_FAILED ) {
    fprintf( stderr, "%s: mmap failed: %s\n", progname, strerror(errno) );
    exit( EXIT_FAILURE );
}

printf( "%s: reading the text: ", progname );
for ( i = 0; i < len; i++ )
    printf( "%c", ptr[i] );
printf( "\n" );

close( fd );
munmap( ptr, len );
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

shm_ctl_special()

Give special attributes to a shared memory object

Synopsis:

```
#include <sys/mman.h>

int shm_ctl_special( int fd,
                    int flags,
                    uint64_t paddr,
                    uint64_t size,
                    unsigned special );
```

Arguments:

fd

The file descriptor that's associated with the shared memory object, as returned by [shm_open\(\)](#) (p. 2935).

flags

One or more of the following bits, defined in `<sys/mman.h>`:

- SHMCTL_ANON — allocate anonymous memory.
- SHMCTL_GLOBAL — a hint that any mapping to the object could be global across all processes.



In order to use the SHMCTL_GLOBAL flag, your process must have the PROCmgr_AID_MEM_GLOBAL ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

- SHMCTL_HIGHUSAGE — a hint that the object is used a lot, and so the system should try to do things to speed up access to it.
- SHMCTL_ISADMA — memory should be suitable for ISA DMA (e.g. it should be below 16 MB, and shouldn't cross 64 KB boundaries).
- SHMCTL_LAZY — delay allocating the memory until it's referenced.

If you create anonymous shared memory objects (by calling [mmap\(\)](#) (p. 1656) with MAP_ANON | MAP_SHARED and a file descriptor of -1), a MAP_LAZY flag implicitly sets the SHMCTL_LAZY flag on the object.

- SHMCTL_LAZYWRITE — a hint that a mapping of this object could use lazy-writing mechanisms.

- `SHMCTL_LOWERPROT` — a hint that the system may map this object in such a way that it trades lower memory protection for better performance.
- `SHMCTL_NODEFRAG` — used with `SHMCTL_PHYS` | `SHMCTL_ANON` to specify that you want physically contiguous memory, but not so much that you're willing to wait for defragmentation. If contiguous memory isn't immediately available, the call gives an error of `ENOMEM`, and you can retry with just `SHMCTL_ANON` if you're willing to accept discontinuity.
- `SHMCTL_NOX64K` — memory shouldn't cross 64 KB boundaries.
- `SHMCTL_PHYS` — use physical address, or allocate physically contiguous memory if used with `SHMCTL_ANON`.



In order to use the `SHMCTL_PHYS` flag, your process must have the `PROC_MGR_AID_MEM_PHYS` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

-
- `SHMCTL_PRIV` — a hint that a mapping of this object may require privileged access.
 - `SHMCTL_REPEAT` — create a strided physical object. For example:

```
shm_ctl( fd, SHMCTL_PHYS, baseaddr, vstride );  
shm_ctl( fd, SHMCTL_REPEAT, pstride, count );
```



Some of the above bits have specific meanings for different processors. For more information, see the documentation for [shm_ctl\(\)](#) (p. 2925).

paddr

A physical address to assign to the object, if you set `SHMCTL_PHYS` in *flags*.

size

The new size of the object, in bytes, regardless of the `ANON/PHYS` flag.

special

Processor-specific flags; see the following sections below:

- [ARM-specific flags](#) (p. 2933)

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `shm_ctl_special()` function modifies the attributes of the shared memory object identified by the handle, `fd`. This handle is the value returned by `shm_open()` (p. 2935).

The `shm_ctl_special()` function is similar to `shm_ctl()` (p. 2925), but has an additional processor-specific argument, `special`.

- In order to call `shm_ctl_special()`, your process must have the `PROCGR_AID_MEM_SPECIAL` ability enabled. For more information, see `procgr_ability()` (p. 2141).
- Calling `shm_ctl_special()` with a `special` argument of 0 **isn't** equivalent to calling `shm_ctl()`. There's an internal flag that tells the memory manager which function you called, so it can treat shared objects differently. Specifying 0 for `special` could clear some special bits that are on by default.
- The combination `SHMCTL_ANON | SHMCTL_PHYS` has the same behavior as for `mmap()` (p. 1656): it indicates that you want physically contiguous RAM to be allocated for the object.
- On ARM targets, once you've called `shm_ctl()` for a shared memory object, you can't resize it. You must unmap and unlink the shared object, and then recreate it.
- If you specify `SHMCTL_PHYS` in the flags, then `paddr` and `size` must be even multiples of the page size (`sysconf(_SC_PAGE_SIZE)`).



ARM-specific flags

For ARM platforms, the `special` argument specifies the page table entry (PTE) bits to be set when the object is mapped.

Returns:

0

Success.

-1

An error occurred (`errno` (p. 539) is set).

Errors:

EAGAIN

The special value doesn't match the current special value.

EBADF

The shared memory object is already “special.”

EINVAL

An invalid combination of flags was specified.

ENOMEM

One of the following:

- There wasn't enough memory.
- You specified SHMCTL_PHYS | SHMCTL_ANON | SHMCTL_NODEFRAG, and contiguous memory wasn't immediately available. You can retry with just SHMCTL_ANON if you're willing to accept discontiguity.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

shm_open()

Open a shared memory object

Synopsis:

```
#include <fcntl.h>
#include <sys/mman.h>

int shm_open( const char * name,
              int oflag,
              mode_t mode );
```

Arguments:

name

The name of the shared memory object that you want to open; see below.

oflag

A combination of the following bits (defined in <fcntl.h>):

- **O_RDONLY** — open for read access only.
- **O_RDWR** — open for read and write access.
- **O_CREAT** — if the shared memory object exists, this flag has no effect, except as noted under **O_EXCL** below. Otherwise, the shared memory object is created, and its permissions are set in accordance with the value of *mode* and the file mode creation mask of the process.
- **O_EXCL** — if **O_EXCL** and **O_CREAT** are set, then *shm_open()* fails if the shared memory segment exists.

The check for the existence of the shared memory object, and the creation of the object if it doesn't exist, are atomic with respect to other processes executing *shm_open()*, naming the same shared memory object with **O_EXCL** and **O_CREAT** set.

- **O_TRUNC** — if the shared memory object exists, and it's successfully opened **O_RDWR**, the object is truncated to zero length and the mode and owner are unchanged.

mode

The permission bits for the memory object are set to the value of *mode*, except those bits set in the process's file creation mask. For more

information, see [umask\(\)](#) (p. 3557), and “[Access permissions](#) (p. 3165)” in the documentation for *stat()*.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *shm_open()* function returns a file descriptor that's associated with the shared “memory object” specified by *name*. This file descriptor is used by other functions to refer to the shared memory object (for example, [mmap\(\)](#) (p. 1656), [mprotect\(\)](#) (p. 1699)). The [FD_CLOEXEC](#) (p. ?) file descriptor flag in *fcntl()* is set for this file descriptor.

The *name* argument is interpreted as follows:

<i>name</i>	Pathname space entry
<i>entry</i>	<i>CWD/entry</i>
<i>/entry</i>	<i>/dev/shmem/entry</i>
<i>entry/newentry</i>	<i>CWD/entry/newentry</i>
<i>/entry/newentry</i>	<i>/entry/newentry</i>

where *CWD* is the current working directory for the program at the point that it calls *shm_open()*.

The state of the shared memory object, including all data associated with it, persists until the shared memory object is unlinked and all other references are gone.

Returns:

A nonnegative integer, which is the lowest numbered unused file descriptor, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

Permission to create the shared memory object is denied.

The shared memory object exists and the permissions specified by *oflag* are denied, or *O_TRUNC* is specified and write permission is denied.

EEXIST

O_CREAT and O_EXCL are set, and the named shared memory object already exists.

EINTR

The *shm_open()* call was interrupted by a signal.

EINVAL

An underlying call to [resmgr_open_bind\(\)](#) (p. 2636) failed.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the *name* argument exceeds NAME_MAX.

ENFILE

Too many shared memory objects are currently open in the system.

ENOENT

O_CREAT isn't set, and the named shared memory object doesn't exist, or O_CREAT is set and either the name prefix doesn't exist or the *name* argument points to an empty string.

ENOSPC

There isn't enough space to create the new shared memory object.

ENOSYS

The *shm_open()* function isn't supported by this implementation.

Examples:

This example sets up a shared memory object, but doesn't really do anything with it:

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <sys/mman.h>

int main( int argc, char** argv )
{
    int fd;
    unsigned* addr;
```

```

/*
 * In case the unlink code isn't executed at the end
 */
if( argc != 1 ) {
    shm_unlink( "/bolts" );
    return EXIT_SUCCESS;
}

/* Create a new memory object */
fd = shm_open( "/bolts", O_RDWR | O_CREAT, 0777 );
if( fd == -1 ) {
    fprintf( stderr, "Open failed:%s\n",
             strerror( errno ) );
    return EXIT_FAILURE;
}

/* Set the memory object's size */
if( ftruncate( fd, sizeof( *addr ) ) == -1 ) {
    fprintf( stderr, "ftruncate: %s\n",
             strerror( errno ) );
    return EXIT_FAILURE;
}

/* Map the memory object */
addr = mmap( 0, sizeof( *addr ),
             PROT_READ | PROT_WRITE,
             MAP_SHARED, fd, 0 );
if( addr == MAP_FAILED ) {
    fprintf( stderr, "mmap failed: %s\n",
             strerror( errno ) );
    return EXIT_FAILURE;
}

printf( "Map addr is 0x%08x\n", addr );

/* Write to shared memory */
*addr = 1;

/*
 * The memory object remains in
 * the system after the close
 */
close( fd );

/*
 * To remove a memory object
 * you must unlink it like a file.
 * This may be done by another process.
 */
shm_unlink( "/bolts" );

return EXIT_SUCCESS;
}

```

This example uses a shared memory object to share data with a forked process:

```

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

main(int argc, char * argv[])
{
    int      fd;
    unsigned *addr;

    /*
     * In case the unlink code isn't executed at the end
     */
    if (argc != 1) {
        shm_unlink("/bolts");
        return EXIT_SUCCESS;
    }

    /* Create a new memory object */

```

```

fd = shm_open("/bolts", O_RDWR | O_CREAT, 0777);
if (fd == -1) {
    fprintf(stderr, "Open failed : %s\n",
        strerror(errno));
    return EXIT_FAILURE;
}

/* Set the memory object's size */
if (ftruncate(fd, sizeof(*addr)) == -1) {
    fprintf(stderr, "ftruncate : %s\n", strerror(errno));
    return EXIT_FAILURE;
}

/* Map the memory object */
addr = mmap(0, sizeof(*addr), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, 0);
if (addr == MAP_FAILED) {
    fprintf(stderr, "mmap failed:%s\n", strerror(errno));
    return EXIT_FAILURE;
}

printf("Map addr is %6.6X\n", addr);
printf("Press break to stop.\n");
sleep(3);    /* So you can read above message */

/*
 * We unlink so object goes away on last close.
 */
shm_unlink("/bolts");

*addr = '0';
if (fork())
    for (;;)
        if (*addr == '0')
            putc(*addr = '1', stderr);
        else
            sched_yield();
else
    for (;;)
        if (*addr == '1')
            putc(*addr = '0', stderr);
        else
            sched_yield();
return EXIT_SUCCESS;
}

```

Classification:

POSIX 1003.1 SHM (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

shm_unlink()

Remove a shared memory object

Synopsis:

```
#include <sys/mman.h>

int shm_unlink( const char * name );
```

Arguments:

name

The name of the shared memory object that you want to remove.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *shm_unlink()* function removes the name of the shared memory object specified by *name*. After removing the name, you can't use [shm_open\(\)](#) (p. 2935) to access the object.

This function doesn't affect any references to the shared memory object (i.e. file descriptors or memory mappings). If more than one reference to the shared memory object exists, then the link count is decremented, but the shared memory segment isn't actually removed until you remove all open and map references to it.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

Permission to unlink the shared memory object is denied.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The length of the *name* argument exceeds NAME_MAX.

ENOENT

The named shared memory object doesn't exist, or the *name* argument points to an empty string.

ENOSYS

The *shm_unlink()* function isn't supported by this implementation.

Examples:

See [shm_open\(\)](#) (p. 2935).

Classification:

[POSIX 1003.1 SHM](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

shutdown()

Shut down part of a full-duplex connection

Synopsis:

```
#include <sys/socket.h>

int shutdown( int s,
              int how );
```

Arguments:

s

A descriptor for the socket, as returned by [socket\(\)](#) (p. 3075).

how

How you want to shut down the connection:

If <i>how</i> is:	The TCP/IP manager won't allow:
SHUT_RD	Further receives
SHUT_WR	Further sends
SHUT_RDWR	Further sends and receives

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The *shutdown()* call shuts down all or part of a full-duplex connection on the socket associated with *s*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid descriptor s.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

shutdown_system()

Shut down the system

Synopsis:

```
#include <sys/shutdown.h>

void shutdown_system( int type,
                     int flags );
```

Arguments:

type

The type of shutdown; one of the following:

- SHUTDOWN_REBOOT—reboot the system.
- SHUTDOWN_SYSTEM—shut down the system.

flags

Zero or more of the following bits:

- FLAG_FAST—do a “fast” shutdown.
- FLAG_DEBUG—debug mode; don't actually shut down.
- FLAG_VERBOSE—be verbose.
- FLAG_VERY_VERBOSE—be very verbose.
- FLAG_NONROOT—ignore root privileges (not used in the default library).
- FLAG_UNATTENDED—don't ask for options (not used in the default library).
- FLAG_NO_DAEMON—don't daemonize the process.

Library:

libshutdown.a

Use the -l shutdown option to gcc to link against this library.

Description:

The *shutdown_system()* function reboots or shuts down the system, depending on the *type* argument. It tries first to shut down any processes.

You can customize this function by providing your own callouts to override those in the library:

- [shutdown_classify\(\)](#) (p. 2945)
- [shutdown_display\(\)](#) (p. 2945)
- [shutdown_done\(\)](#) (p. 2946)
- [shutdown_error\(\)](#) (p. 2946)
- [shutdown_process\(\)](#) (p. 2946)
- [shutdown_progress\(\)](#) (p. 2946)
- [shutdown_prompt\(\)](#) (p. 2946)

This library is used by shutdown, with callouts provided by the utility.

shutdown_classify() callout

```
ProcessClass_t shutdown_classify( ProcessInfo_t const *pip );
```

This callout determines whether an application is console-based, based on the information provided in the structure that *pip* points to. The `ProcessInfo_t` structure is defined as follows:

```
typedef struct
{
    uint64_t start_time;
    pid_t pid;
    int8_t class;
    uint8_t padding[3];
    char *name;
} ProcessInfo_t;
```

The `shutdown_system()` function recognizes some applications by their name or other information, and calls `shutdown_classify()` to determine the type of applications that it doesn't recognize.

The `shutdown_classify()` callout should return one of the defined class types:

- `CLASS_UNKNOWN`
- `CLASS_APP`—an application.
- `CLASS_DAEMON`—a server, resource manager, or driver.
- `CLASS_FSYS`—a filesystem.
- `CLASS_DISPLAY`—a display driver.

The default stub unconditionally returns `CLASS_UNKNOWN`.

shutdown_display() callout

```
void shutdown_display( int type,
                      DisplayData_t const *display);
```

This callout can display the name of a process or class of processes being shut down, depending on the *type* argument:

- `DISPLAY_CLASS`—a new class of processes was encountered.
- `DISPLAY_PROC`—a new process was encountered.

The default stub does nothing.

The `DisplayData_t` structure provides information about what's currently being shut down. It's defined as follows:

```
typedef union
{
    int proc_class;          /* use for DISPLAY_CLASS */
    char const *proc_name;   /* use for DISPLAY_PROC */
} DisplayData_t;
```

The `proc_class` member is one of the class types that [`shutdown\_classify\(\)`](#) (p. 2945) returns.

`shutdown_done()` callout

```
void shutdown_done( int type );
```

This callout is called before shutting down the display processes, but after shutting down the rest. The `type` argument is the typed passed to `shutdown_system()`.

The default stub does nothing.

`shutdown_error()` callout

```
void shutdown_error( char const *msg );
```

The library invokes this callout if an error has occurred; the library exits, indicating a failure, after issuing this callout. The default stub does nothing.

`shutdown_process()` callout

```
void shutdown_process(void);
```

This callout is called to process intermediate events during the shutdown. The default stub does nothing.

`shutdown_progress()` callout

```
void shutdown_progress( int done,
                        int total );
```

This callout is called to display the progress being made at shutting down. The `done` argument indicates the number of processes already shut down, out of the given `total`.

The default stub does nothing.

`shutdown_prompt()` callout

```
int shutdown_prompt(char const *name, pid_t pid);
```

This callout is called to deal with a process that isn't responding. It should return one of the following to indicate to the library how to deal with the errant process:

- `PROMPT_KILL`—immediately raise a `SIGKILL` signal for the process.
- `PROMPT_WAIT`—continue to wait for the pending `SIGTERM`.
- `PROMPT_SKIP`—ignore this process, and shut down other processes.
- `PROMPT_CANCEL`—abort the shutdown in progress.

The default stub returns PROMPT_SKIP.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

sigaction()

Examine or specify the action associated with a signal

Synopsis:

```
#include <signal.h>

int sigaction( int sig,
               const struct sigaction * act,
               struct sigaction * oact );
```

Arguments:

sig

The signal number (defined in <signal.h>). For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

act

NULL, or a pointer to a `sigaction` structure that specifies how you want to modify the action for the given signal. For more information about this structure, see below.

oact

NULL, or a pointer to a `sigaction` structure that the function fills with information about the current action for the signal.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use *sigaction()* to examine or specify (or both) the action that's associated with a specific signal:

- If *act* isn't NULL, the specified signal is modified.
- If *oact* isn't NULL, the previous action is stored in the structure it points to.



In order to attach signal handlers to a process with a different real or effective user ID, your process must have the `PROC_MGR_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The structure `sigaction` contains the following members:

`void (*sa_handler)();`

Address of a signal handler or action for nonqueued signals.

`void (*sa_sigaction)(int signo, siginfo_t* info, void* other);`

Address of a signal handler or action for queued signals.

`sigset_t sa_mask`

An additional set of signals to be masked (blocked) during execution of the signal-catching function.

`int sa_flags`

Special flags to affect behavior of the signal:

- `SA_NOCLDSTOP` is only used when the signal is `SIGCHLD`. It tells the system not to generate a `SIGCHLD` on the parent for children who stop via `SIGSTOP`.
- `SA_SIGINFO` tells the OS to queue this signal. The default is not to queue a signal delivered to a process. If a signal isn't queued, setting the same signal multiple times on a process or thread before it runs results in only the last signal's being delivered. If you set `SA_SIGINFO`, the signals are queued and they're all delivered.

The `sa_handler` and `sa_sigaction` members of `act` are implemented as a union and share common storage. They differ only in their prototypes, with `sa_handler` being used for POSIX 1003.1a signals and `sa_sigaction` being used for POSIX 1003.1b queued realtime signals. The values stored using either name can be one of:

function

The address of a signal catching function. See below for details.

`SIG_DFL`

Use the default action for the signal:

- `SIGIO`, `SIGURG`, and `SIGWINCH` — ignore the signal.
- `SIGCHLD` — ignore the signal, but still let the process's children become zombies.

- SIGSTOP — stop the process.
- SIGCONT — continue the program.
- all other signals — kill the process.

SIG_IGN

Ignore the signal. Setting SIG_IGN for a signal that's pending discards all pending signals, whether it's blocked or not. New signals are discarded. If you set the action for SIGCHLD to SIG_IGN, your process's children won't enter the zombie state, and the process won't be able to use wait on their death using *wait()* or *waitpid()*.

The *function* member of *sa_handler* or *sa_sigaction* is always invoked with the following arguments:

```
void handler(int signo, siginfo_t *info, void *other)
```

If you have an old-style signal handler of the form:

```
void handler(int signo)
```

the extra arguments are still placed by the kernel, but the function simply ignores them.

While in the handler, *signo* is masked, preventing nested signals of the same type. In addition, any signals set in the *sa_mask* member of *act* are also ORed into the mask. When the handler returns through a normal return, the previous mask is restored and any pending and now unmasked signals are acted on. You return to the point in the program where it was interrupted. If the thread was blocked in the kernel when the interruption occurred, the kernel call returns with an EINTR (see [ChannelCreate\(\)](#) (p. 278) and [SyncMutexLock\(\)](#) (p. 3294) for exceptions to this).



It isn't safe to use floating-point operations in signal handlers.

The *siginfo_t* structure of the *function* in *sa_handler* or *sa_sigaction* contains at least the following members:

int si_signo

The signal number, which should match the *signo* argument to the handler.

int si_code

A signal code, provided by the generator of the signal:

- SI_USER — the [kill\(\)](#) (p. 1494) function generated the signal.
- SI_QUEUE — the [sigqueue\(\)](#) (p. 3007) function generated the signal.
- SI_TIMER — a timer generated the signal.

- SI_ASYNCIO — asynchronous I/O generated the signal.
- SI_MESGQ — POSIX (not QNX Neutrino) messages queues generated the signal.

union signal si_value

A value associated with the signal, provided by the generator of the signal.

You can't ignore or catch SIGKILL or SIGSTOP.

Signal handlers and actions are defined for the process and affect all threads in the process. For example, if one thread ignores a signal, then all threads ignore the signal.

You can target a signal at a thread, process, or process group (see [SignalKill\(\)](#) (p. 2986)). When targeted at a process, at most one thread receives the signal. This thread must have the signal unblocked (see [SignalProcmask\(\)](#) (p. 2991)) to be a candidate for receiving it. All synchronously generated signals (e.g. SIGSEGV) are always delivered to the thread that caused them.



If you use [longjmp\(\)](#) (p. 1549) to return from a signal handler, the signal remains masked. You can use [siglongjmp\(\)](#) (p. 2970) to restore the mask to the state saved by a previous call to [sigsetjmp\(\)](#) (p. 3010).

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

Insufficient system resources are available to set up the signal's action.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

The signal *signo* isn't valid.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main( void )
{
    extern void handler();
    struct sigaction act;
    sigset_t set;

    sigemptyset( &set );
    sigaddset( &set, SIGUSR1 );
    sigaddset( &set, SIGUSR2 );

    /*
     * Define a handler for SIGUSR1 such that when
     * entered both SIGUSR1 and SIGUSR2 are masked.
     */
    act.sa_flags = 0;
    act.sa_mask = set;
    act.sa_handler = &handler;
    sigaction( SIGUSR1, &act, NULL );

    kill( getpid(), SIGUSR1 );

    /* Program will terminate with a SIGUSR2 */
    return EXIT_SUCCESS;
}

void handler( signo )
{
    static int first = 1;

    if( first ) {
        first = 0;
        kill( getpid(), SIGUSR1 ); /* Prove signal masked */
        kill( getpid(), SIGUSR2 ); /* Prove signal masked */
    }
}

/*
 * - SIGUSR1 is set from main(), handler() is called.
 * - SIGUSR1 and SIGUSR2 are set from handler().
 * - however, signals are masked until we return to main().
 * - returning to main() unmask SIGUSR1 and SIGUSR2.
 * - pending SIGUSR1 now occurs, handler() is called.
 * - pending SIGUSR2 now occurs. Since we don't have
 *   a handler for SIGUSR2, we are killed.
 */
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigaddset()

Add a signal to a set

Synopsis:

```
#include <signal.h>

int sigaddset( sigset_t *set,
               int signo );
```

Arguments:

set

A pointer to the `sigset_t` object that you want to add the signal to.

signo

The signal that you want to add. For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *sigaddset()* function adds *signo* to the set pointed to by *set*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The signal *signo* isn't valid.

Examples:

See [sigemptyset\(\)](#) (p. 2959).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

sigblock()

Add to the mask of signals to block

Synopsis:

```
#include <unix.h>

int sigblock( int mask );
```

Arguments:

mask

A bitmask of the signals that you want to block.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigblock()* function adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signals are blocked if the appropriate bit in *mask* is a 1; the macro [sigmask\(\)](#) (p. 2972) is provided to construct the mask for a given signum. The *sigblock()* returns the previous mask. You can restore the previous mask by calling [sigsetmask\(\)](#) (p. 3012).

In normal usage, a signal is blocked using *sigblock()*. To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there's no work to be done, and the process pauses awaiting work by using [sigpause\(\)](#) (p. 3000) with the mask returned by *sigblock()*.

It isn't possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

Returns:

The previous set of masked signals.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

Caveats:

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

sigdelset()

Delete a signal from a set

Synopsis:

```
#include <signal.h>

int sigdelset( sigset_t *set,
               int signo );
```

Arguments:

set

A pointer to the `sigset_t` object that you want to remove the signal from.

signo

The signal that you want to remove. For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *sigdelset()* function deletes *signo* from the set pointed to by *set*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The signal *signo* isn't valid.

Examples:

See [sigemptyset\(\)](#) (p. 2959).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

sigemptyset()

Initialize a set to contain no signals

Synopsis:

```
#include <signal.h>

int sigemptyset( sigset_t *set );
```

Arguments:

set

A pointer to the `sigset_t` object that you want to initialize.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigemptyset()* function initializes *set* to contain no signals.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void print( sigset_t set, int signo )
{
    printf( "Set %8.8lx. Signal %d is ", set, signo );
    if( sigismember( &set, signo ) )
        printf( "a member.\n" );
    else
        printf( "not a member.\n" );
}
```

```
int main( void )
{
    sigset_t set;

    sigemptyset( &set );
    print( set, SIGINT );

    sigfillset( &set );
    print( set, SIGINT );

    sigdelset( &set, SIGINT );
    print( set, SIGINT );

    sigaddset( &set, SIGINT );
    print( set, SIGINT );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

sigevent

Structure that describes an event

Synopsis:

The `sigevent` structure, which is defined in `<sys/siginfo.h>`, is complicated; see below.

Description:

This structure describes an event. The `int sigev_notify` member indicates how the notification is to occur, as well as which of the other members are used:

<i>sigev_notify</i>	<i>sigev_signo</i>	<i>sigev_coid</i>	<i>sigev_priority</i>	<i>sigev_code</i>	<i>sigev_value</i>
SIGEV_INTR (p. 2962)					
SIGEV_NONE (p. 2962)					
SIGEV_PULSE (p. 2962)		Connection	Priority	Code	Value
SIGEV_SIGNAL (p. 2962)	Signal				
SIGEV_SIGNAL_CODE (p. 2963)	Signal			Code	Value
SIGEV_SIGNAL_THREAD (p. 2963)	Signal			Code	Value
SIGEV_THREAD (p. 2964) (special — see below)					Value
SIGEV_UNBLOCK (p. 2964)					

The `<sys/siginfo.h>` file also defines some macros to make initializing the `sigevent` structure easier. All the macros take a pointer to a `sigevent` structure as their first argument, *event*, and set the *sigev_notify* member to the appropriate value.

- These macros are QNX Neutrino extensions.
- The library might use hidden bits in the *sigev_notify* member of the `sigevent` structure. If you ever test the value of this member after creating a `sigevent`, don't compare it directly to a value; use the `SIGEV_GET_TYPE()` macro. For example, instead of:



```
if( my_event.sigev_notify == SIGEV_PULSE)
```

use:

```
if( SIGEV_GET_TYPE(&my_event) == SIGEV_PULSE)
```

SIGEV_INTR

Send an interrupt notification to a specific thread. No other fields in the structure are used.

The initialization macro is:

SIGEV_INTR_INIT(&event)

SIGEV_NONE

Don't send any notification. No other fields in the structure are used.

The initialization macro is:

SIGEV_NONE_INIT(&event)

SIGEV_PULSE

Send a pulse. The following fields are used:

int *sigev_coid*

The connection ID. This should be attached to the channel with which the pulse will be received.

short *sigev_priority*

The priority of the pulse.

If you want the thread that receives the pulse to run at the initial priority of the process, set *sigev_priority* to *SIGEV_PULSE_PRIO_INHERIT*.

short *sigev_code*

A code to be interpreted by the pulse handler. Although *sigev_code* can be any 8-bit signed value, you should avoid *sigev_code* values less than zero in order to avoid conflict with kernel or pulse codes generated by a QNX Neutrino manager. These codes all start with *_PULSE_CODE_* and are defined in *<sys/neutrino.h>*; for more information, see the documentation for the [_pulse](#) (p. 2440) structure. A safe range of pulse values is *_PULSE_CODE_MINAVAIL* to *_PULSE_CODE_MAXAVAIL*.

void **sigev_value.sival_int*

A 32-bit value to be interpreted by the pulse handler.

The initialization macro is:

SIGEV_PULSE_INIT(&event, coid, priority, code, value)

SIGEV_SIGNAL

Send a signal to a process. The following fields are used:

int sigev_signo

The signal to raise. This must be in the range from 1 through NSIG – 1.

The initialization macro is:

SIGEV_SIGNAL_INIT(&event, signal)

If you need to set the *sigev_value* for a SIGEV_SIGNAL event (for example if SA_SIGINFO is set), you can use this macro:

SIGEV_SIGNAL_VALUE_INIT(&event, signal, value)

SIGEV_SIGNAL_CODE

This is similar to SIGEV_SIGNAL, except that SIGEV_SIGNAL_CODE also includes a value and a code. The following fields are used:

int sigev_signo

The signal to raise. This must be in the range from 1 through NSIG – 1.

short sigev_code

A code to be interpreted by the signal handler. This must be in the range from SI_MINAVAIL through SI_MAXAVAIL.

void *sigev_value.sival_int

A 32-bit value to be interpreted by the signal handler.

The initialization macro is:

SIGEV_SIGNAL_CODE_INIT(&event, signal, value, code)

SIGEV_SIGNAL_THREAD

Send a signal to a specific thread, depending on the situation:

- If used with *MsgSend*()*, *MsgReceive*()*, and *MsgDeliverEvent()* (p. 1736), the thread is the one identified by the *rcvid* that *MsgReceive*()* returns.
- For timers, it's the thread that called *timer_settime()* (p. 3439).
- For *InterruptAttach()* (p. 1147) and *InterruptAttachEvent()* (p. 1154), it's the thread attaching to the interrupt.
- For *SyncMutexEvent()* (p. 3291), it's the thread that called the function.

In the case of timers, *SyncMutexEvent()*, and interrupts, if the thread dies before the event gets delivered, the kernel sends the signal to a random thread in the same process.

The following fields are used:

int sigev_signo

The signal to raise. This must be in the range from 1 through `NSIG - 1`.

short *sigev_code*

A code to be interpreted by the signal handler. This must be in the range from `SI_MINAVAIL` through `SI_MAXAVAIL`.

void **sigev_value.sival_int*

A 32-bit value to be interpreted by the signal handler.

The initialization macro is:

`SIGEV_SIGNAL_THREAD_INIT( &event, signal, value, code )`

SIGEV_THREAD

Create a new thread.



We don't recommend using this type of event. Pulses are more efficient.

The following fields are used:

void (sigev_notify_function*)(union *sigval*)**

A pointer to the function to be notified.

pthread_attr **sigev_notify_attributes*

A pointer to thread attributes. This must be `NULL`, or point to a structure initialized by [*pthread_attr_init\(\)*](#) (p. 2213) at the time of delivery.

void **sigev_value.sival_ptr*

A value that's to be passed to the notification function.

The initialization macro is:

`SIGEV_THREAD_INIT( &event, fn, value, attr )`

The `sigval` union is defined as follows:

```
union sigval {
    int      sival_int;
    void     * sival_ptr;
};
```

SIGEV_UNBLOCK

Force a thread to become unblocked. No other fields in the structure are used.

The initialization macro is:

`SIGEV_UNBLOCK_INIT( &event )`

Critical threads

If you're using adaptive partitioning, you can use a `sigevent` to make a thread run as critical or not.



This feature was added in the QNX Neutrino Core OS 6.3.2. For more information, see the Adaptive Partitioning *User's Guide*.

After setting up the `sigevent` structure as appropriate, use these macros to set or clear the hidden bit that makes a thread run as critical or not:

SIGEV_MAKE_CRITICAL(&event)

Make the targeted thread run as critical.

SIGEV_CLEAR_CRITICAL(&event)

Make the targeted thread not run as critical.

The receiving thread doesn't have to do anything to make itself critical or noncritical; the adaptive partitioning scheduler does this automatically.



These macros use hidden bits in the `sigev_notify` member of the `sigevent` structure. Don't compare this member directly to a value; use the `SIGEV_GET_TYPE()` macro instead, as described above.

Classification:

[QNX Neutrino](#) (p. 46)

sigfillset()

Initialize a set to contain all signals

Synopsis:

```
#include <signal.h>
int sigfillset( sigset_t *set );
```

Arguments:

set

A pointer to the sigset_t object that you want to initialize.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *sigfillset()* function initializes *set* to contain all signals.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Examples:

See [sigemptyset\(\)](#) (p. 2959).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

sigismember()

See if a given signal is in a given set

Synopsis:

```
#include <signal.h>

int sigismember( const sigset_t *set,
                  int signo );
```

Arguments:

set

A pointer to the `sigset_t` object that you want to check.

signo

The signal that you want to check for membership in the set. For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for `SignalAction()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigismember()` function tests if `signo` is in the set pointed to by `set`.

Returns:

1

The `signo` is in the set.

0

The `signo` isn't in the set.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The signal *signo* isn't valid.

Examples:

See [sigemptyset\(\)](#) (p. 2959).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

siglongjmp()

Restore the environment saved by sigsetjmp(), including the signal mask

Synopsis:

```
#include <setjmp.h>

void siglongjmp( sigjmp_buf env,
                 int val );
```

Arguments:

env

The environment saved by the most recent call to [sigsetjmp\(\)](#) (p. 3010).

val

The value that you want *sigsetjmp()* to return.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *siglongjmp()* function is a superset of the *longjmp()* function, but also restores the thread's saved signal mask if (and only if) one was saved in the *env* argument by a previous call to *sigsetjmp()*.



*Don't use [longjmp\(\)](#) (p. 1549) or *siglongjmp()* to restore an environment saved by a call to [setjmp\(\)](#) (p. 2875) or [sigsetjmp\(\)](#) (p. 3010) in another thread. If you're lucky, your application will crash; if not, it'll look as if it works for a while, until random scribbling on the stack causes it to crash.*

Returns:

The same values that [longjmp\(\)](#) (p. 1549) returns.

Examples:

See [longjmp\(\)](#) (p. 1549).

Classification:*POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigmask()

Construct a mask for a signal number

Synopsis:

```
#include <unix.h>

#define sigmask(s)      (1L<<((s)-1))
```

Arguments:

s

The signal that you want to create a mask for. For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

This macro constructs the mask for a given signal number. Use *sigmask()* in conjunction with [sigblock\(\)](#) (p. 2955), [sigsetmask\(\)](#) (p. 3012), and [sigunblock\(\)](#) (p. 3018).

Returns:

The signal mask.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

signal()

Set handling for exceptional conditions

Synopsis:

```
#include <signal.h>

void ( * signal( int sig,
                void ( * func)(int) ) )( int );
```

Arguments:

sig

The signal number (defined in `<signal.h>`). For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

func

The function that you want to call when the signal is raised.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *signal()* function is used to specify an action to take place when certain conditions are detected while a program executes. See the `<signal.h>` header file for definitions of these conditions, and also refer to the *System Architecture* manual.



In order to attach signal handlers to a process with a different real or effective user ID, your process must have the `PROCmgr_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

There are three types of actions that can be associated with a signal: `SIG_DFL`, `SIG_IGN` or a pointer to a function. Initially, all signals are set to `SIG_DFL` or `SIG_IGN` prior to entry of the [main\(\)](#) (p. 1566) routine. An action can be specified for each of the conditions, depending upon the value of the *func* argument, as discussed below.

***func* is a function**

When *func* is a function name, that function is called in a manner equivalent to the following code sequence:

```
/* "sig_no" is condition being signalled */
signal( sig_no, SIG_DFL );
(*func)( sig_no );
```

The *func* function may do the following:

- Return.
- Terminate the program by calling `_exit()` (p. 596) or `abort()` (p. 54).
- Call `longjmp()` (p. 1549) or `siglongjmp()` (p. 2970). If you use `longjmp()` to return from a signal handler, the signal remains masked. You can use `siglongjmp()` to restore the mask to the state saved in a previous call to `sigsetjmp()`.



It isn't safe to use floating-point operations in signal handlers.

After returning from the signal-catching function, the receiving process resumes execution at the point at which it was interrupted.

The signal catching function is described as follows:

```
void func( int sig_no )
{
    ...
}
```

It isn't possible to catch the SIGSTOP or SIGKILL signals.

Since signal-catching functions are invoked asynchronously with process execution, you need to take into account the same sort of things that you would in a multithreaded program when inspecting or manipulating shared resources.

***func* is SIG_DFL**

If *func* is SIG_DFL, the default action for the condition is taken.

If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal is generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag (see [sigaction\(\)](#) (p. 2948)). While a process is stopped, any additional signals that are sent to the process aren't delivered until the process is continued, except SIGKILL, which always terminates the receiving process.

Setting a signal action to SIG_DFL for a signal that is pending, and whose default action is to ignore the signal (for example, SIGCHLD), causes the pending signal to be discarded, whether or not it's blocked.

***func* is SIG_IGN**

If *func* is SIG_IGN, the indicated condition is ignored.

You can't set the action for the SIGSTOP and SIGKILL signals to SIG_IGN.

Setting a signal action to SIG_IGN for a signal that's pending causes the pending signal to be discarded, whether or not it is blocked.

If a process sets the action for the SIGCHLD signal to SIG_IGN, its children won't enter the zombie state and the process can't use [wait\(\)](#) (p. 3652) or [waitpid\(\)](#) (p. 3666) to wait on their deaths.

Handling a condition

When a condition is detected, it may be handled by a program, it may be ignored, or it may be handled by the usual default action (often causing an error message to be printed on the *stderr* stream followed by program termination).

A condition can be generated by a program using the [raise\(\)](#) (p. 2482) or [kill\(\)](#) (p. 1494) function

Returns:

The previous value of *func* for the indicated condition, or SIG_ERR if the request couldn't be handled ([errno](#) (p. 539) is set to EINVAL).

Examples:

```
#include <stdlib.h>
#include <signal.h>

sig_atomic_t signal_count;

void MyHandler( int sig_number )
{
    ++signal_count;
}

int main( void )
{
    signal( SIGFPE, MyHandler ); /* set own handler */
    signal( SIGABRT, SIG_DFL ); /* Default action */
    signal( SIGFPE, SIG_IGN ); /* Ignore condition */
    return (EXIT_SUCCESS);
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SignalAction(), SignalAction_r()

Examine and/or specify actions for signals

Synopsis:

```
#include <sys/neutrino.h>

int SignalAction( pid_t pid,
                  void ( * sigstub)(),
                  int signo,
                  const struct sigaction * act,
                  struct sigaction * oact );

int SignalAction_r( pid_t pid,
                    void * (sigstub)(),
                    int signo,
                    const struct sigaction * act,
                    struct sigaction * oact );
```

Arguments:

pid

A process ID, or 0 for the current process.

sigstub

The address of a signal stub handler. This is a small piece of code in the user's space that interfaces the user's signal handler to the kernel. The library provides a standard one, `__signalstub()`. This argument can be NULL if *act* is also NULL.

signo

The signal whose action you want to set or get; see “[POSIX signals](#) (p. 2978),” below.

act

NULL, or a pointer to a `sigaction` structure that specifies the new action for the signal. For more information, see “[Signal actions](#) (p. 2981),” below.

oact

NULL, or a pointer to a `sigaction` structure where the function can store the old action.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *SignalAction()* and *SignalAction_r()* kernel calls let the calling process examine or specify (or both) the action to be associated with a specific signal in the process *pid*. If *pid* is zero, the calling process is used. The argument *signo* specifies the signal.



- You should call the POSIX [sigaction\(\)](#) (p. 2948) function or the ANSI [signal\(\)](#) (p. 2974) function instead of using these kernel calls directly.
- In order to attach signal handlers to a process with a different real or effective user ID, your process must have the PROC_MGR_AID_SIGNAL ability enabled. For more information, see [proc_mgr_ability\(\)](#) (p. 2141).

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2984) section for details.

POSIX signals

The signals are defined in <signal.h>, and so are these global variables:

char * const sys_siglist[]

An array of signal names.

int sys_nsig

The number of entries in the *sys_siglist* array.

As mentioned earlier, the OS defines a total of 64 signals. Their range is as follows:

Signal range	Description
1 ... 57	57 POSIX signals (including traditional UNIX signals)
41 ... 56	16 POSIX realtime signals (SIGRTMIN to SIGRTMAX)
57 ... 64	Eight special-purpose QNX Neutrino signals

The POSIX and UNIX signals include:

SIGHUP

Hangup.

SIGINT

Interrupt.

SIGQUIT

Quit.

SIGILL

Illegal instruction (not reset when caught).

One possible cause for this signal is trying to perform an operation that requires *I/O privileges*. A thread can request these privileges by making sure it has the `PROCmgr_AID_IO` ability enabled (see [procmgr_ability\(\)](#) (p. 2141)) and then calling [ThreadCtl\(\)](#) (p. 3408), specifying the `_NT0_TCTL_IO` flag:

```
ThreadCtl( _NT0_TCTL_IO, 0 );
```

SIGTRAP

Trace trap (not reset when caught).

SIGIOT

IOT instruction.

SIGABRT

Used by [abort\(\)](#) (p. 54).

SIGDEADLK

Mutex deadlock; see [SyncMutexEvent\(\)](#) (p. 3291).



SIGDEADLK and SIGEMT refer to the same signal.

SIGEMT

EMT instruction (emulation trap).

SIGFPE

Floating point exception.

SIGKILL

Kill (can't be caught or ignored)

SIGBUS

Bus error.

SIGSEGV

Segmentation violation.

SIGSYS

Bad argument to system call.

SIGPIPE

Write on pipe with no reader.

SIGALRM

Realtime alarm clock.

SIGTERM

Software termination signal from kill.

SIGUSR1

User-defined signal 1.

SIGUSR2

User-defined signal 2.

SIGCHLD

Death of child.

SIGPWR

Power-fail restart.

SIGWINCH

Window change.

SIGURG

Urgent condition on I/O channel.

SIGPOLL

System V name for SIGIO.

SIGIO

Asynchronous I/O.

SIGSTOP

Sendable stop signal not from tty.

SIGTSTP

Stop signal from tty.

SIGCONT

Make a stopped process continue.

SIGTTIN

Attempted background tty read.

SIGTTOU

Attempted background tty write.



You can't ignore or catch SIGCONT, SIGKILL, or SIGSTOP.

There are 16 POSIX 1003.1b realtime signals, including:

SIGRTMIN

First realtime signal.

SIGRTMAX

Last realtime signal.

The entire range of signals goes from _SIGMIN (1) to _SIGMAX (64).

QNX Neutrino signals

QNX Neutrino uses the top eight signals for special purposes. They're always masked, and attempts to unmask them are ignored. They include named signals (e.g., SIGSELECT) and unnamed ones.

Signal actions

If *act* isn't NULL, then the specified signal is modified. If *oact* isn't NULL, the previous action is stored in the structure it points to. You can use various combinations of *act* and *oact* to query or set (or both) the action for a signal.

The structure `sigaction` contains the following members:

```
void (*sa_handler)();
```

The address of a signal handler or action for nonqueued signals.

```
void (*sa_sigaction) (int signo, siginfo_t *info, void *other);
```

The address of a signal handler or action for queued signals.

```
sigset_t sa_mask
```

An additional set of signals to be masked (blocked) during execution of the signal-catching function.

int *sa_flags*

Special flags that affect the behavior of the signal:

- **SA_NOCLDSTOP** — don't generate a **SIGCHLD** on the parent for children who stop via **SIGSTOP**. This flag is used only when the signal is **SIGCHLD**.
- **SA_SIGINFO** — queue this signal. The default is not to queue a signal delivered to a process. If a signal isn't queued, and the same signal is set multiple times on a process or thread before it runs, only the last signal is delivered. If you set the **SA_SIGINFO** flag, the signals are queued, and they're all delivered.

The *sa_handler* and *sa_sigaction* members of *act* are implemented as a union, and share common storage. They differ only in their prototype, with *sa_handler* being used for POSIX 1003.1a signals, and *sa_sigaction* being used for POSIX 1003.1b queued realtime signals. The values stored using either name can be one of:

function

The address of a signal-catching function. See below for details.

SIG_DFL

Use the default action for the signal:

- **SIGIO**, **SIGURG**, and **SIGWINCH** — ignore the signal.
- **SIGCHLD** — ignore the signal, but still let the process's children become zombies.
- **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU** — stop the process.
- **SIGCONT** — make the program continue.
- all other signals — kill the process.

SIG_IGN

Ignore the signal. Setting **SIG_IGN** for a signal that's pending discards all pending signals, whether it's blocked or not. New signals are discarded. If you set the action for **SIGCHLD** to **SIG_IGN**, your process's children won't enter the zombie state, and the process won't be able to use [wait\(\)](#) (p. 3652) or [waitpid\(\)](#) (p. 3666) to wait on their deaths.

The *function* member of *sa_handler* or *sa_sigaction* is always invoked with the following arguments:

```
void handler(int signo, siginfo_t* info, void* other)
```

If you have an old-style signal handler of the form:

```
void handler(int signo)
```

the microkernel passes the extra arguments, but the function simply ignores them.

While in the handler, *signo* is masked, preventing nested signals of the same type. In addition, any signals set in the *sa_mask* member of *act* are also ORed into the mask. When the handler returns through a normal return, the previous mask is restored, and any pending and now unmasked signals are acted on. You return to the point in the program where it was interrupted. If the thread was blocked in the kernel when the interruption occurred, the kernel call returns with an EINTR (see [ChannelCreate\(\)](#) (p. 278) and [SyncMutexLock\(\)](#) (p. 3294) for exceptions to this).



It isn't safe to use floating-point operations in signal handlers.

When you specify a handler, you must provide the address of a signal stub handler for *sigstub*. This is a small piece of code in the user's space that interfaces the user's signal handler to the kernel. The library provides a standard one, *__signalstub()*.

The *siginfo_t* structure of the *function* in *sa_handler* or *sa_sigaction* contains at least the following members:

int *si_signo*

The signal number, which should match the *signo* argument to the handler.

int *si_code*

A signal code, provided by the generator of the signal:

- SI_USER — the [kill\(\)](#) (p. 1494) function generated the signal.
- SI_QUEUE — the [sigqueue\(\)](#) (p. 3007) function generated the signal.
- SI_TIMER — a timer generated the signal.
- SI_ASYNCIO — asynchronous I/O generated the signal.
- SI_MSGQ — POSIX (not QNX Neutrino) messages queues generated the signal.

union *sigval si_value*

A value associated with the signal, provided by the generator of the signal.

Signal handlers and actions are defined for the process and affect all threads in the process. For example, if one thread ignores a signal, then all threads ignore the signal.

You can target a signal at a thread, process or process group (see [SignalKill\(\)](#) (p. 2986)). When targeted at a process, at most one thread receives the signal. This thread must have the signal unblocked (see [SignalProcmask\(\)](#) (p. 2991)) to be a candidate for receiving

it. All synchronously generated signals (e.g. SIGSEGV) are always delivered to the thread that caused them.

In a multithreaded process, if a signal terminates a thread, by default all threads and thus the process are terminated. You can override this standard POSIX behavior when you create the thread; see [ThreadCreate\(\)](#) (p. 3402).



If you use [longjmp\(\)](#) (p. 1549) to return from a signal handler, the signal remains masked. You can use [siglongjmp\(\)](#) (p. 2970) to restore the mask to the state saved by a previous call to [sigsetjmp\(\)](#) (p. 3010).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SignalAction()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

SignalAction_r()

EOK is returned on success. This function does **NOT** set [errno](#). If an error occurs, any value in the Errors section may be returned.

Errors:

EAGAIN

The system was unable to allocate a signal handler. This indicated critically low memory.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The value of *signo* is less than 1 or greater than _SIGMAX, or you tried to set SIGKILL or SIGSTOP to something other than SIG_DFL.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process indicated by *pid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SignalKill(), SignalKill_r()

Send a signal to a process group, process, or thread

Synopsis:

```
#include <sys/neutrino.h>

int SignalKill( uint32_t nd,
                pid_t pid,
                int tid,
                int signo,
                int code,
                int value );

int SignalKill_r( uint32_t nd,
                  pid_t pid,
                  int tid,
                  int signo,
                  int code,
                  int value );
```

Arguments:

nd

The node descriptor of the node on which to look for *pid* and *tid*. To search the local node, set *nd* to ND_LOCAL_NODE or 0.

pid

0, or the ID of the process to send the signal to; see below.

tid

0, or the ID of the thread to send the signal to; see below.

signo

The signal that you want to send. There are a total of 64 signals available. Of these, at least 8 are POSIX realtime signals that range from SIGRTMIN to SIGRTMAX. For a complete list of signals, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*. Valid user signals range from 1 to (NSIG - 1).

You can OR the SIG_APPID flag into the *signo* argument. This tells *SignalKill()* to send the signal to all the processes with an application ID that matches the *pid* argument.

code, value

The code and value associated with the signal; see [SignalAction\(\)](#) (p. 2977).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SignalKill()* and *SignalKill_r()* kernel calls send the signal *signo* with a code specified by *code* and a value specified by *value* to a process group, process, or thread.



In order to send signals to a process with a different real or effective user ID, your process must have the `PROCmgr_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2989) section for details.

If *signo* is zero, no signal is sent, but the validity of *pid* and *tid* are checked. You can use this as a test for existence.

SignalKill() implements the capabilities of the POSIX functions [kill\(\)](#) (p. 1494), [sigqueue\(\)](#) (p. 3007), and [pthread_kill\(\)](#) (p. 2310) in one call. Consider using these functions instead of invoking these kernel calls directly.

The *pid* and *tid* determine the target of the signal, as follows:

<i>pid</i>	<i>tid</i>	target
= 0	—	Hit the process group of the caller
< 0	—	Hit a process group identified by <i>-pid</i>
> 0	= 0	Hit a single process identified by <i>pid</i>
> 0, and you've ORed <code>SIG_APPID</code> into the <i>signo</i> argument	—	Hit all processes with the application ID identified by <i>pid</i>
> 0	> 0	Hit a single thread in process <i>pid</i> identified by <i>tid</i>

If the target is a thread, the signal is always delivered to exactly that thread. If the thread has the signal blocked — see [SignalProcmask\(\)](#) (p. 2991) — the signal remains pending on the thread.

If the target is a process, the signal is delivered to a thread that has the signal unblocked; see [SignalProcmask\(\)](#), [SignalSuspend\(\)](#) (p. 2995), and [SignalWaitinfo\(\)](#) (p. 2997). If multiple threads have the signal unblocked, only one thread is given the signal. Which thread receives the signal isn't deterministic. To make it deterministic, you can:

- Have all threads except one block all signals; that thread handles all signals.

Or:

- Target the signals to specific threads.

If all threads have the signal blocked, it's made pending on the process. The first thread to unblock the signal receives the pending signal. If a signal is pending on a thread, it's never retargetted to the process or another thread, regardless of changes to the signal-blocked mask.

If the target is a process group, the signal is delivered as above to each process in the group.

A multithreaded application typically has one thread responsible for catching most or all signals. Threads that don't wish to be directly involved with signals block all signals in their mask.

The signal-blocked mask is maintained on a per-thread basis. The signal-ignore mask and signal handlers are maintained at the process level and are shared by all threads.

If multiple signals are delivered before the target can run and process the signals, the system queues them in priority order if the `SA_SIGINFO` bit was set for *signo*. Lower numbered signals have greater priority. If the `SA_SIGINFO` bit isn't set for *signo*, then at most one signal is queued at any time. Additional signals with the same *signo* replace existing ones. This is the default behavior for POSIX signal handlers installed using the old [signal\(\)](#) (p. 2974) function. The newer [sigaction\(\)](#) (p. 2948) function lets you control queuing or not on a per-signal basis. Signals with a *code* of `SI_TIMER` are never queued.

The *code* and *value* are always saved with the signal. This allows you to deliver data with the signal whether or not `SA_SIGINFO` has been set on the *signo*. If `SA_SIGINFO` is set, you can use signals to deliver small amounts of data without loss. If you wish to pass significant data, you may wish to consider using [MsgSendPulse\(\)](#) (p. 1789) and [MsgSendv\(\)](#) (p. 1801), which deliver data with much greater efficiency.

When a thread receives a signal by a signal handler or [SignalWaitinfo\(\)](#) call, it can retrieve the *signo*, *code* and *value* from a `siginfo_t` structure, which contains at least the following members:

int *si_signo*

The signal number.

int *si_code*

The signal code.

union *sigval* *si_value*

The signal value.

The value of *si_code* is limited to an 8-bit signed value as follows:

Value	Description
$-128 \leq si\_code \leq 0$	User values
$0 < signo \leq 127$	System values generated by the kernel

Some of the common user values defined by POSIX are:

- **SI_USER** — the [kill\(\)](#) (p. 1494) function generated the signal.
- **SI_QUEUE** — the [sigqueue\(\)](#) (p. 3007) function generated the signal.
- **SI_TIMER** — a timer generated the signal.
- **SI_ASYNCIO** — asynchronous I/O generated the signal.
- **SI_MESGQ** — POSIX (not QNX Neutrino) messages queues generated the signal.

A successful return from this function means the signal has been delivered. What the process(es) or thread does with the signal isn't considered.

If a thread delivers signals that the receiving process has marked as queued faster than the receiver can consume them, the kernel may fail the call if it runs out of signal queue entries. If the *signo*, *code*, and *value* don't change, the kernel performs signal compression by saving an 8-bit count with each queued signal.

Blocking states

None. In the network case, lower priority threads may run.

Returns:

The only difference between these functions is the way they indicate errors:

SignalKill()

If an error occurs, -1 is returned and sets [errno](#) (p. 539). Any other value returned indicates success.

SignalKill_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:**EAGAIN**

The kernel had insufficient resources to enqueue the signal.

EINVAL

The value of *signo* is less than 0 or greater than (`_NSIG - 1`).

ENOREMOTE

The node descriptor is for a remote node, but the network manager isn't running.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process or process group indicated by *pid* or thread indicated by *tid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SignalProcmask(), SignalProcmask_r()

Modify or examine the signal-blocked mask of a thread

Synopsis:

```
#include <sys/neutrino.h>

int SignalProcmask( pid_t pid,
                    int tid,
                    int how,
                    const sigset_t* set,
                    sigset_t* oldset );

int SignalProcmask_r( pid_t pid,
                     int tid,
                     int how,
                     const sigset_t* set,
                     sigset_t* oldset );
```

Arguments:

pid

0, or a process ID; see below.

tid

0, or a thread ID; see below.

how

The manner in which you want to change the set:

- **SIG_BLOCK** — the resulting set is the union of the current set and the signal set pointed to by *set*.
- **SIG_UNBLOCK** — the resulting set is the intersection of the current set and the signal set pointed to by *set*.
- **SIG_SETMASK** — the resulting set is the signal set pointed to by *set*.

As a special case, you can use the *how* argument to query the current set of pending signals:

- **SIG_PENDING** — the combined set of pending signals on the thread and process are saved in the signal set pointed to by *oldset*. The *set* argument is ignored.

set

NULL, or a pointer to a `sigset_t` object that specified the set of signals to be used to change the currently blocked set.

oldset

NULL, or a pointer to a `sigset_t` object where the function can store the previous blocked mask.

You can use various combinations of *set* and *oldset* to query or change (or both) the signal-blocked mask for a signal.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls modify or examine the signal-blocked mask of the thread *tid* in process *pid*. If *pid* is zero, the current process is assumed. If *tid* is zero, *pid* is ignored and the calling thread is used.

The *SignalProcmask()* and *SignalProcmask_r()* functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 2993) section for details.



- Instead of using these kernel calls directly, consider calling [pthread_sigmask\(\)](#) (p. 2407) or [sigprocmask\(\)](#) (p. 3004).
- In order to set the signal-blocked mask for a process with a different real or effective user ID, your process must have the `PROCmgr_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

When a signal is unmasked, the kernel checks for pending signals on the thread and, if there aren't any pending, checks for pending signals on the process:

Check	Action
Signal pending on thread	The signal is immediately acted upon.
Signal pending on process	The signal is moved to the thread and is immediately acted upon.
No signal pending	No signal action performed until delivery of an unblocked signal.

It isn't possible to block the SIGCONT, SIGKILL, or SIGSTOP signals. It also isn't possible to unmask the special QNX Neutrino signals (SIGSELECT and the unnamed ones).

When a signal handler is invoked, the signal responsible is automatically masked before its handler is called; see [SignalAction\(\)](#) (p. 2977). If the handler returns normally, the operating system restores the signal mask present just before the handler was called as an atomic operation. Changes made using *SignalProcmask()* in the handler are undone.

When a signal is targeted at a process, the kernel delivers it to at most one thread (see [SignalKill\(\)](#) (p. 2986)) that has the signal unblocked. If multiple threads have the signal unblocked, only one thread is given the signal. Which thread receives the signal isn't deterministic. To make it deterministic, you can:

- Have all threads except one block all signals; that thread handles all signals.

Or:

- Target signals to specific threads.

If all threads have the signal blocked, it's made pending on the process. The first thread to unblock the signal receives the pending signal. If a signal is pending on a thread, it's never retargetted to the process or another thread, regardless of changes to the signal-blocked mask.

Signals targeted at a thread always affect that thread alone.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SignalProcmask()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

SignalProcmask_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EAGAIN

The system was unable to allocate a signal handler. This indicates critically low memory.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The value of *how* is invalid, or you tried to set SIGKILL or SIGSTOP to something other than SIG_DFL.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process indicated by *pid* or thread indicated by *tid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SignalSuspend(), SignalSuspend_r()

Suspend a thread until a signal is received

Synopsis:

```
#include <sys/neutrino.h>

int SignalSuspend( const sigset_t* set );
int SignalSuspend_r( const sigset_t* set );
```

Arguments:

set

A pointer to a `sigset_t` object that specifies the signals you want to wait for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls replace the thread's signal mask with the set of signals pointed to by *set* and then suspends the thread until delivery of a signal whose action is either to execute a signal-catching function (then return), or to terminate the thread. On return, the previous signal mask is restored.

The *SignalSuspend()* and *SignalSuspend_r()* functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 2996) section for details.



Instead of using these kernel calls directly, consider calling [sigsuspend\(\)](#) (p. 3014).

Attempts to block `SIGKILL` or `SIGSTOP` are ignored. This is done without causing an error.

If you're using *SignalSuspend()* to synchronously wait for a signal, consider using the more efficient POSIX 1003.1b realtime [sigwaitinfo\(\)](#) (p. 3022) call.

Blocking states

STATE_SIGSUSPEND

The calling thread blocks waiting for a signal.

Returns:

The only difference between these functions is the way they indicate errors.
Since *SignalSuspend()* and *SignalSuspend_r()* block until interrupted, there's no successful return value.

SignalSuspend()

-1 is always returned and *errno* (p. 539) is set.

SignalSuspend_r()

errno is **NOT** set, a value in the Errors section is returned.

If the signal handler calls *longjmp()* (p. 1549) or *siglongjmp()* (p. 2970), *SignalSuspend()* and *SignalSuspend_r()* don't return.

Errors:

EINTR

The call was interrupted by a signal (this is the normal error).

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

ETIMEDOUT

A kernel timeout unblocked the call. See *TimerTimeout()* (p. 3464).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SignalWaitinfo(), SignalWaitinfo_r()

Select a pending signal

Synopsis:

```
#include <sys/neutrino.h>

int SignalWaitinfo( const sigset_t* set,
                   siginfo_t* info );

int SignalWaitinfo_r( const sigset_t* set,
                     siginfo_t* info );
```

Arguments:

set

A pointer to a `sigset_t` object that specifies the signals you want to wait for.

info

NULL, or a pointer to a `siginfo_t` structure where the function can store information about the signal.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SignalWaitinfo()` and `SignalWaitinfo_r()` kernel calls select the pending signal from the set specified by *set*. If no signal in *set* is pending at the time of the call, the thread blocks until one or more signals in *set* become pending or until interrupted by an unblocked, caught signal.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 2998) section for details.



Instead of using these kernel calls directly, consider calling `sigwaitinfo()` (p. 3022).

If the *info* argument isn't NULL, information on the selected signal is stored there as follows:

siginfo_t member	Description
<i>si_signo</i>	Selected signal number
<i>si_code</i>	Signal code
<i>si_value</i>	Signal value

If, while *SignalWaitinfo()* is waiting, a caught signal occurs that isn't blocked, the signal handler is invoked and *SignalWaitinfo()* is interrupted with an error of EINTR.

Blocking states

STATE_SIGWAITINFO

The calling thread blocks waiting for a signal.

Returns:

The only difference between these functions is the way they indicate errors:

SignalWaitinfo()

A signal number. If an error occurs, -1 is returned and *errno* (p. 539) is set.

SignalWaitinfo_r()

A signal number. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EINTR

The call was interrupted by a signal.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

ETIMEDOUT

A kernel timeout unblocked the call. See *TimerTimeout()* (p. 3464).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

sigpause()

Wait for a signal

Synopsis:

```
#include <signal.h>

int sigpause( int sig );
```

Arguments:

sig

A mask containing the signal number that you want to wait for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigpause()* function assigns *sig* to the set of masked signals and then waits for a signal to arrive; on return, the set of masked signals is restored. The *mask* argument is usually 0 to indicate that no signals are now to be blocked. This function always terminates by being interrupted, returning -1, and setting *errno* to EINTR.

In normal usage, a signal is blocked using *sigblock()* (p. 2955). To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigpause()* with the mask returned by *sigblock()*.

It isn't possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

Returns:

-1; *errno* (p. 539) is set to EINTR.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

sigpending()

Examine the set of pending, masked signals for a process

Synopsis:

```
#include <signal.h>

int sigpending( sigset_t *set );
```

Arguments:

set

A pointer to a `sigset_t` object that the function sets to indicate the pending, masked signals.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *sigpending()* function is used to examine the set of pending signals that are masked (blocked) from delivery to the calling thread and that are pending on the calling process or thread. They're saved in the signal set pointed to by *set*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

A fault occurred while accessing the buffer pointed to by *set*.

Examples:

See [sigprocmask\(\)](#) (p. 3004).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigprocmask()

Examine or change the signal mask for a thread

Synopsis:

```
#include <signal.h>

int sigprocmask( int how,
                  const sigset_t *set,
                  sigset_t *oset );
```

Arguments:

how

The manner in which you want to change the set:

- **SIG_BLOCK** — add the signals pointed to by *set* to the thread mask.
- **SIG_UNBLOCK** — remove the signals pointed to by *set* from the thread mask.
- **SIG_SETMASK** — set the thread mask to be the signals pointed to by *set*.

set

NULL, or a pointer to a `sigset_t` object that defines the signals that you want to change in the thread's signal mask. If this argument is NULL, the *how* argument is ignored.

oset

NULL, or a pointer to a `sigset_t` object that the function sets to indicate the thread's current signal mask.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *sigprocmask()* function is used to examine or change (or both) the signal mask for the calling thread. If the value of *set* isn't NULL, it points to a set of signals to be used to change the currently blocked set.



In order to set the signal-blocked mask for a process with a different real or effective user ID, your process must have the `PROCGR_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The *set* argument isn't changed. The resulting set is maintained in the process table of the calling thread. If a signal occurs on a signal that's masked, it becomes pending, but doesn't affect the execution of the process. You can examine pending signals by calling [sigpending\(\)](#) (p. 3002). When a pending signal is unmasked, it's acted upon immediately, before this function returns.

When a signal handler is invoked, the signal responsible is automatically masked before its handler is called. If the handler returns normally, the operating system restores the signal mask present just before the handler was called as an atomic operation. Changes made using *sigprocmask()* in the handler are undone.

The *sigaction()* function lets you specify any mask that's applied before a handler is invoked. This can simplify multiple signal handler design.



To maintain code portability, use *pthread_sigmask()* in a multithreaded application.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

Insufficient system resources are available to mask the signals.

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

The value of *how* is invalid, or you tried to set SIGKILL or SIGSTOP to something other than SIG_DFL.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main( void )
{
    sigset_t set, oset, pset;

    sigemptyset( &set );
    sigaddset( &set, SIGINT );
    sigprocmask( SIG_BLOCK, &set, &oset );
    printf( "Old set was %8.8ld.\n", oset );

    sigpending( &pset );
    printf( "Pending set is %8.8ld.\n", pset );

    kill( getpid(), SIGINT );

    sigpending( &pset );
    printf( "Pending set is %8.8ld.\n", pset );

    sigprocmask( SIG_UNBLOCK, &set, &oset );

    /* The program terminates with a SIGINT */
    return( EXIT_SUCCESS );
}
```

Classification:

[POSIX 1003.1 CX](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigqueue()

Queue a signal to a process

Synopsis:

```
#include <signal.h>

int sigqueue ( pid_t pid,
               int signo,
               const union sigval value );
```

Arguments:

pid

The ID of the process that you want to signal.

signo

Zero, or the number of the signal that you want to queue for the process. For more information, see “[POSIX signals](#) (p. 2978)” in the documentation for *SignalAction()*.

value

The value to queue with the signal.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigqueue()* function causes the signal, *signo* to be sent with the specified value to the process, *pid*. If *signo* is zero, error checking is performed, but no signal is actually sent. This is one way of checking to see if *pid* is valid.

The condition required for a process to have permission to queue a signal to another process is the same as for the [kill\(\)](#) (p. 1494) function:

- the real or effective user ID of the sending process must match the real or effective user ID of the receiving process

Or:

- the calling process must have the `PROCMGR_AID_SIGNAL` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The `sigqueue()` function returns immediately. If `SA_SIGINFO` is set for *signo* and if the resources are available to queue the signal, the signal is queued and sent to the receiving process. If `SA_SIGINFO` isn't set for the *signo*, then *signo* is sent to the receiving process if the signal isn't already pending.

If *pid* causes *signo* to be generated for the sending process, and if *signo* isn't blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a `sigwait()` function for *signo*, then either *signo* or at least one pending unblocked signal is delivered to the calling thread before `sigqueue()` returns.

Should any of multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected for delivery, the lowest numbered one is delivered. The selection order between realtime and nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:

EAGAIN

No resources were available to queue the signal. The process has already queued the maximum number of signals as returned by:

```
sysconf( _SC_SIGQUEUE_MAX )
```

that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.

EINVAL

The value of the *signo* argument is an invalid or unsupported signal number.

ENOSYS

The function `sigqueue()` isn't supported by this implementation.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process *pid* doesn't exist.

Classification:

[POSIX 1003.1 RTS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigsetjmp()

Save the environment, including the signal mask

Synopsis:

```
#include <setjmp.h>

int sigsetjmp( sigjmp_buf env,
               int savemask );
```

Arguments:

env

A buffer where the function can save the calling environment.

savemask

Nonzero if you want to save the process's current signal mask, otherwise 0.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigsetjmp()* function behaves in the same way as the [setjmp\(\)](#) (p. 2875) function when *savemask* is zero. If *savemask* is nonzero, then *sigsetjmp()* also saves the thread's current signal mask as part of the calling environment.



Don't use [longjmp\(\)](#) (p. 1549) or [siglongjmp\(\)](#) (p. 2970) to restore an environment saved by a call to [setjmp\(\)](#) (p. 2875) or *sigsetjmp()* in another thread. If you're lucky, your application will crash; if not, it'll look as if it works for a while, until random scribbling on the stack causes it to crash.

Returns:

Zero on the first call, or nonzero if the return is the result of a call to [siglongjmp\(\)](#) (p. 2970).

Examples:

See [setjmp\(\)](#) (p. 2875).

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigsetmask()

Set the mask of signals to block

Synopsis:

```
#include <unix.h>

int sigsetmask( int mask );
```

Arguments:

mask

A bitmask of the signals that you want to block.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *sigsetmask()* function sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in *mask* is a 1; the macro [sigmask\(\)](#) (p. 2972) is provided to construct the mask for a given signum.

In normal usage, a signal is blocked using [sigblock\(\)](#) (p. 2955). To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using [sigpause\(\)](#) (p. 3000) with the mask returned by [sigblock\(\)](#).

It isn't possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

Returns:

The previous set of masked signals.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

sigsuspend()

Replace the signal mask, and then suspend the thread

Synopsis:

```
#include <signal.h>

int sigsuspend( const sigset_t *sigmask );
```

Arguments:

sigmask

A pointer to a `sigset_t` object that specifies the signals that you want in the thread's signal mask.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigsuspend()* function replaces the thread's signal mask with the set of signals pointed to by *sigmask* and then suspends the thread until delivery of a signal whose action is either to execute a signal-catching function (then return), or to terminate the thread.

Returns:

-1 (if the function returns); [errno](#) (p. 539) is set.

Errors:

EFAULT

A fault occurred trying to access the buffers provided.

EINTR

A signal was caught by the calling thread, and control is returned from the signal-catching function.

Examples:

```

/*
 * This program pauses until a signal other than
 * a SIGINT occurs. In this case a SIGALRM.
 */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

sigset_t    set;

int main( void )
{
    sigemptyset( &set );
    sigaddset( &set, SIGINT );

    printf( "Program suspended and immune to breaks.\n" );
    printf( "A SIGALRM will terminate the program"
           " in 10 seconds.\n" );
    alarm( 10 );
    sigsuspend( &set );
    return( EXIT_SUCCESS );
}

```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigtimedwait()

Wait for a signal or a timeout

Synopsis:

```
#include <signal.h>

int sigtimedwait( const sigset_t *set,
                  siginfo_t *info,
                  const struct timespec *timeout );
```

Arguments:

set

A set of signals from which the function selects a pending signal.

info

NULL, or a pointer to a `siginfo_t` structure where the function can store information about the signal.

timeout

NULL, or a pointer to a [timespec](#) (p. 3473) structure that specifies the maximum time to wait for a pending signal.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigtimedwait()` function selects a pending signal from `set`, atomically clears it from the set of pending signals in the process, and returns that signal number.

If the `info` argument isn't NULL, `sigtimedwait()` stores the selected signal in the `si_signo` member of `info`, and the cause of the signal in the `si_code` member.

If any value is queued to the selected signal, the first queued value is dequeued and, if the `info` argument is non-NULL, the value is stored in the `si_value` member of `info`. The system resources used to queue the signal are released and made available to queue other signals. If no value is queued, the content of the `si_value` member is undefined.

If no further signals are queued for the selected signal, the pending indication for that signal is reset.

If none of the signals specified by *set* are pending, *sigtimedwait()* waits for the time interval specified by the *timespec* structure *timeout*. The `CLOCK_MONOTONIC` clock is used to measure the time interval.

If *timeout* is zero and if none of the signals specified by *set* are pending, then *sigtimedwait()* returns immediately with an error. If *timeout* is `NULL`, *sigtimedwait()* behaves the same as [sigwaitinfo\(\)](#) (p. 3022).

Returns:

The selected signal number, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The *timeout* expired before a signal specified in *set* was generated, or all kernel timers are in use.

EFAULT

A fault occurred while accessing the provided buffers.

EINTR

The wait was interrupted by an unblocked, caught signal.

EINVAL

The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1000 million or *set* contains an invalid or unsupported signal number.

Classification:

[POSIX 1003.1 RTS](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigunblock()

Unblock signals

Synopsis:

```
#include <unix.h>

int sigunblock( int mask );
```

Arguments:

mask

A bitmask of the signals that you want to unblock.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigunblock()* function removes the signals specified in *mask* from the set of signals currently being blocked from delivery. Signals are unblocked if the appropriate bit in *mask* is a 1; the macro [sigmask\(\)](#) (p. 2972) is provided to construct the mask for a given signal. The *sigunblock()* returns the previous mask. You can restore the previous mask by calling [sigsetmask\(\)](#) (p. 3012).

In normal usage, a signal is blocked using [sigblock\(\)](#) (p. 2955). To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using [sigpause\(\)](#) (p. 3000) with the mask returned by *sigblock()*.

It isn't possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

Returns:

The previous set of masked signals.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

sigwait()

Wait for a pending signal

Synopsis:

```
#include <signal.h>

int sigwait( const sigset_t *set,
             int *sig );
```

Arguments:

set

A pointer to a `sigset_t` object that specifies the signals you want to wait for.

sig

A pointer to a location where the function can store the signal that it cleared.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigwait()* function selects a pending signal from *set*, atomically clears it from the set of pending signals in the system, and returns that signal number in *sig*. If there are multiple signals queued for the signal number selected, the first signal causes a return from *sigwait()* and the rest remain queued. If no signal in *set* is pending at the time of the call, the thread is suspended until one or more becomes pending.

The signals defined by *set* must be blocked before you call *sigwait()*; otherwise, the behavior is undefined. The effect of *sigwait()* on the signal actions for the signals in *set* is unspecified.

If more than one thread is using *sigwait()* to wait for the same signal, only one of the threads returns from *sigwait()* with the signal number — which one is unspecified.

Returns:

0

Success (that is, one of the signals specified by set is pending or has been generated).

EINTR

The *sigwait()* function was interrupted by a signal.

EINVAL

The *set* argument contains an invalid or unsupported signal number.

EFAULT

A fault occurred while accessing the provided buffers.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sigwaitinfo()

Wait for a specified signal and return its information

Synopsis:

```
#include <signal.h>

int sigwaitinfo ( const sigset_t *set,
                  siginfo_t *info );
```

Arguments:

set

A pointer to a `sigset_t` object that specifies the signals you want to wait for.

info

NULL, or a pointer to a `siginfo_t` structure where the function can store information about the signal.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sigwaitinfo()* function selects a pending signal from *set*, atomically clears it from the set of pending signals in the system, and returns that signal number.

If there's no pending signal, *sigwaitinfo()* blocks until the specified signal is pending. If the specified signal is already pending, upon call to *sigwaitinfo()*, the call immediately returns without blocking.

If *info* isn't NULL, *sigwaitinfo()* stores the selected signal in the *si_signo* member of *info*, and the cause of the signal in the *si_code* member.

If any value is queued to the selected signal, the first queued value is dequeued and, if the *info* argument is non-NULL, the value is stored in the *si_value* member of *info*. The system resources used to queue the signal are released and made available to queue other signals. If no value is queued, the content of the *si_value* member is undefined.

If no further signals are queued for the selected signal, the pending indication for that signal is reset.

Returns:

A signal number, or -1 if an error occurred ([errno](#) (p. 539) is set).

Errors:**EFAULT**

A fault occurred while accessing the buffers.

EINTR

The wait was interrupted by an unblocked, caught signal.

Classification:

[POSIX 1003.1 RTS](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sin(), sinf(), sinl()

Calculate the sine of an angle

Synopsis:

```
#include <math.h>

double sin( double x );
float sinf( float x );
long double sinl( long double x );
```

Arguments:

x

The angle, in radians, for which you want to compute the sine.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the sine (specified in radians) of *x*. An argument with a large magnitude may yield a result with little or no significance.

Returns:

The sine value. If *x* is `+/-Inf`, these functions return NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", sin(.5) );
}
```

```
    return( EXIT_SUCCESS );  
}
```

produces the output:

```
0.479426
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

sinh(), sinh(), sinh()

Compute the hyperbolic sine

Synopsis:

```
#include <math.h>

double sinh( double x );
float sinh( float x );
long double asinh( long double x );
```

Arguments:

x

The angle, in radians, for which you want to compute the hyperbolic sine.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

These functions compute the hyperbolic sine (specified in radians) of *x*. A range error occurs if the magnitude of *x* is too large.

Returns:

The hyperbolic sine value.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", sinh(.5) );
}
```

```
    return( EXIT_SUCCESS );  
}
```

produces the output:

```
0.521095
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

sleep()

Suspend a thread for a given length of time

Synopsis:

```
#include <unistd.h>

unsigned int sleep( unsigned int seconds );
```

Arguments:

seconds

The number of realtime seconds that you want to suspend the thread for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sleep()* function suspends the calling thread until the number of realtime seconds specified by the *seconds* argument have elapsed, or the thread receives a signal whose action is either to terminate the process or to call a signal handler.



This function can interfere with the kernel's efforts to manage power usage. The suspension time may be greater than the requested amount, due to the nature of time measurement (see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*), or due to the scheduling of other, higher priority threads by the system.

Returns:

0 if the full time specified was completed; otherwise, the number of seconds unslept if interrupted by a signal.

Examples:

```
/*
 * The following program sleeps for the
 * number of seconds specified in argv[1].
 */
#include <stdlib.h>
#include <unistd.h>
```



```
int main( int argc, char **argv )
{
    unsigned seconds;

    seconds = (unsigned) strtol( argv[1], NULL, 0 );
    sleep( seconds );

    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_sleepon_broadcast()

Wake up multiple threads

Synopsis:

```
#include <pthread.h>

int _sleepon_broadcast( sleepon_t * l,
                       const volatile void * addr );
```

Arguments:

l

A pointer to a `sleepon_t` that you created by calling [_sleepon_init\(\)](#) (p. 3034).

addr

The handle that the threads are waiting on. The value of *addr* is typically a data structure that controls a resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [_sleepon_signal\(\)](#) (p. 3038) and [_sleepon_broadcast\(\)](#) functions are very similar:

- [_sleepon_signal\(\)](#) wakes up a single thread that's waiting on the key, *addr*.
- [_sleepon_broadcast\(\)](#) wakes up all threads that are waiting on the key, *addr*.

The waiting threads must have used the same `sleepon`, *l* and key, *addr*, in order to be woken up.

To be woken up, the calling threads must have been locked by [_sleepon_lock\(\)](#) (p. 3036).

Returns:

0

Success.

0

Failure.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sleepon_destroy()

Destroy a sleepon

Synopsis:

```
#include <pthread.h>

int _sleepon_destroy( sleepon_t * l );
```

Arguments:

l

A pointer to a `sleepon_t` that you created by calling [_sleepon_init\(\)](#) (p. 3034).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_destroy()` function destroys a `sleepon_t` structure, *l*, that has been previously initialized by [_sleepon_init\(\)](#) (p. 3034).

If *l* hasn't been locked by [_sleepon_lock\(\)](#) (p. 3036), `_sleepon_destroy()` locks it before destroying it.

The `sleepon` structure is reference-counted such that, if other threads are blocked waiting for a condition, they're be signaled to wake up, and the last one to wake up frees the memory allocated to the `sleepon`.

Returns:

0

Success.

0

Failure

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sleepon_init()

Initialize a sleepon

Synopsis:

```
#include <pthread.h>

int _sleepon_init( sleepon_t ** pl,
                  unsigned flags);
```

Arguments:

pl

The address of a location where the function can store a pointer to the `sleepon_t` object that it creates.

flags

There are currently no flags defined; pass zero for this argument.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_init()` function allocates a `sleepon_t` object (which is an opaque data structure) and stores a pointer to it in the location that *pl* points to.

Returns:

0

Success.

0

Failure.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sleepon_lock()

Lock a sleepon

Synopsis:

```
#include <pthread.h>

int _sleepon_lock( sleepon_t * l );
```

Arguments:

l

A pointer to a `sleepon_t` that you created by calling [_sleepon_init\(\)](#) (p. 3034).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_lock()` function locks the mutex associated with the `sleepon` structure, *l*.

You must call this function before calling [_sleepon_wait\(\)](#) (p. 3042), [_sleepon_signal\(\)](#) (p. 3038), or [_sleepon_broadcast\(\)](#) (p. 3030).

Returns:

EOK

Success.

EAGAIN

Insufficient system resources were available to lock the mutex.

EDEADLK

The calling thread already owns *mutex*, and the mutex doesn't allow recursive behavior.

EINVAL

Invalid mutex.

The `_sleepon_lock()` function returns the same values as [pthread_mutex_lock\(\)](#) (p. 2318).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sleepon_signal()

Wake up a single thread

Synopsis:

```
#include <pthread.h>

int _sleepon_signal( sleepon_t * l,
                    const volatile void * addr );
```

Arguments:

l

A pointer to a `sleepon_t` that you created by calling [_sleepon_init\(\)](#) (p. 3034).

addr

The handle that the thread is waiting on. The value of *addr* is typically a data structure that controls a resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_signal()` and [_sleepon_broadcast\(\)](#) (p. 3030) functions are very similar:

- `_sleepon_signal()` wakes up a single thread that's waiting on the key, *addr*.
- `_sleepon_broadcast()` wakes up all threads that are waiting on the key, *addr*.

The waiting threads must have used the same `sleepon`, *l* and key, *addr*, in order to be woken up.

To be woken up, the calling threads must have been locked by [_sleepon_lock\(\)](#) (p. 3036).

Returns:

0

Success.

0

Failure.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sleepon_unlock()

Unlock a sleepon

Synopsis:

```
#include <pthread.h>

int _sleepon_unlock( sleepon_t * l );
```

Arguments:

l

A pointer to a `sleepon_t` that you created by calling [_sleepon_init\(\)](#) (p. 3034).

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_unlock()` function unlocks the mutex associated with the `sleepon` structure, *l*. You must have previously locked the mutex by calling [_sleepon_lock\(\)](#) (p. 3036).

Returns:

EOK

Success.

EINVAL

Invalid mutex *mutex*.

EPERM

The current thread doesn't own *mutex*.

The `_sleepon_unlock()` function returns the same values as [pthread_mutex_unlock\(\)](#) (p. 2329).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

_sleepon_wait()

Wait on a sleepon lock

Synopsis:

```
#include <pthread.h>

int _sleepon_wait( sleepon_t * l,
                  const volatile void * addr,
                  uint64_t nsec);
```

Arguments:

l

A pointer to a `sleepon_t` that you created by calling [_sleepon_init\(\)](#) (p. 3034).

addr

The handle that you want to wait on. The value of *addr* is typically a data structure that controls a resource.

nsec

Zero, or the amount of time, in nanoseconds, to wait before the thread wakes up. If this timeout occurs, `ETIMEDOUT` is returned.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_wait()` function blocks on the sleepon *l* using the key *addr* until woken up by either a [_sleepon_signal\(\)](#) (p. 3038) or a [_sleepon_broadcast\(\)](#) (p. 3030) call that uses the same *addr* key.

The calling thread must first have locked the sleepon by calling [_sleepon_lock\(\)](#) (p. 3036).

When the thread returns from this function, it must release the sleepon lock by calling [_sleepon_unlock\(\)](#) (p. 3040).

Returns:

0

Success.

0

Failure; a nonzero *errno* (p. 539) value.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

slog2c()

Log a constant string in a slog2 buffer

Synopsis:

```
#include <slog2.h>

int slog2c( slog2_buffer_t buffer,
            _Uint16t code,
            _Uint8t severity,
            const char* data );
```

Arguments:

buffer

The handle of the buffer you want to log the message in, or **NULL** to use the default buffer that you specified earlier with [*slog2_set_default_buffer\(\)*](#) (p. 3057).

code

A user-specified code that you want to be associated with the message. The slog2 system doesn't interpret this code in any way.

severity

The severity level of this log item; one of:

- SLOG2_SHUTDOWN
- SLOG2_CRITICAL
- SLOG2_ERROR
- SLOG2_WARNING
- SLOG2_NOTICE
- SLOG2_INFO
- SLOG2_DEBUG1
- SLOG2_DEBUG2

For more information, see the entry for [*slog2_register\(\)*](#) (p. 3052).

data

A pointer to the constant string that you want to log in the buffer.

Library:

lib slog2

Use the -l slog2 option to qcc to link against this library.

Description:

The *slog2c()* function logs a constant string to the slog2 buffer.

Returns:

0 on success, or -1 if an error occurred.

Examples:

See [slog2_register\(\)](#) (p. 3052).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

slog2f()

Log a printf-style string in a slog2 buffer

Synopsis:

```
#include <slog2.h>

int slog2f( slog2_buffer_t buffer,
            uint16_t code,
            uint8_t severity,
            const char* format, ... );
```

Arguments:

buffer

The handle of the buffer you want to log the message in, or **NULL** to use the default buffer that you specified earlier with [*slog2_set_default_buffer\(\)*](#) (p. 3057).

code

A user-specified code that you want to be associated with the message. The *slog2* system doesn't interpret this code in any way.

severity

The severity level of this log item; one of:

- **SLOG2_SHUTDOWN**
- **SLOG2_CRITICAL**
- **SLOG2_ERROR**
- **SLOG2_WARNING**
- **SLOG2_NOTICE**
- **SLOG2_INFO**
- **SLOG2_DEBUG1**
- **SLOG2_DEBUG2**

For more information, see the entry for [*slog2_register\(\)*](#) (p. 3052).

format

A [*printf*](#) (p. 2133)-formatted character string used to define the log, followed by any additional arguments, as required.

Library:

lib slog2

Use the -l slog2 option to qcc to link against this library.

Description:

The *slog2f()* function logs a printf-style ASCII-text string in the specified slog2 buffer.

The *vslog2f()* (p. 3628) function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of <stdarg.h>.

Returns:

0 on success, or -1 if an error occurred.

Examples:

See *slog2_register()* (p. 3052).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

Don't use `double` or `float` arguments in an interrupt handler, unless your code is compiled to use software floating-point emulation.

slog2fa()

Log a printf-style string in a slog2 buffer

Synopsis:

```
#include <slog2.h>

int slog2fa( slog2_buffer_t buffer,
             uint16_t code,
             uint8_t severity,
             const char* format, ... );
```

Arguments:

buffer

The handle of the buffer you want to log the message in, or **NULL** to use the default buffer that you specified earlier with [*slog2_set_default_buffer\(\)*](#) (p. 3057).

code

A user-specified code that you want to be associated with the message. The slog2 system doesn't interpret this code in any way.

severity

The severity level of this log item; one of:

- SLOG2_SHUTDOWN
- SLOG2_CRITICAL
- SLOG2_ERROR
- SLOG2_WARNING
- SLOG2_NOTICE
- SLOG2_INFO
- SLOG2_DEBUG1
- SLOG2_DEBUG2

For more information, see the entry for [*slog2_register\(\)*](#) (p. 3052).

format

A [*printf*](#) (p. 2133)-formatted character string used to define the log message.

Additional arguments

An argument vector that specifies the arguments for the format string. You must use the appropriate `SLOG2_FA_*`() helper macro for parameters, as described below. The last parameter in the vector must be `SLOG2_FA_END`.

Library:

`lib slog2`

Use the `-l slog2` option to `qcc` to link against this library.

Description:

The `slog2fa()` function a *printf*-style format string along with its arguments specified in the argument vector. This function uses special formatting macros to ensure that `va_args()` doesn't trigger any interrupts. The format string isn't decoded before writing; it must be decoded during the time of the read.



If you're specifying a string with `%s`, the size must include the NULL character.

You can use the following macros to build the parameters to `slog2fa`:

- `SLOG2_FA_UNSIGNED( x )`
- `SLOG2_FA_SIGNED( x )`
- `SLOG2_FA_CHAR( x )`
- `SLOG2_FA_FLOAT( x )`
- `SLOG2_FA_DOUBLE( x )`
- `SLOG2_FA_STRING( x )`
- `SLOG2_FA_END` — you must end the list with this macro

For example:

```
int8_t some_number = -1;
slog2fa( ..., "string:%s, some_number:%d", SLOG2_FA_STRING( "Hello world" ),
        SLOG2_FA_SIGNED( some_number ),
        SLOG2_FA_END );
```

Returns:

0 on success, or -1 if an error occurred.

Examples:

See [slog2_register\(\)](#) (p. 3052).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

Don't use `double` or `float` arguments in an interrupt handler, unless your code is compiled to use software floating-point emulation.

slog2_get_verbosity()

Get the verbosity level of the entire buffer set

Synopsis:

```
#include <slog2.h>

uint8_t slog2_get_verbosity( slog2_buffer_t buffer );
```

Arguments:

buffer

A handle for a buffer in the buffer set whose verbosity you want to get, or NULL if you want to get the verbosity of the default buffer that you earlier set with [slog2_set_default_buffer\(\)](#) (p. 3057).

Library:

lib slog2

Use the -l slog2 option to qcc to link against this library.

Description:

The *slog2_get_verbosity()* function returns the verbosity of the *entire* buffer set that the given buffer belongs to.

Returns:

The current verbosity level (see the entry for [slog2_register\(\)](#) (p. 3052)), or SLOG2_INVALID_VERBOSITY if *buffer* is NULL and there's currently no default buffer.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slog2_register()

Register and allocate a new slog2 buffer set

Synopsis:

```
#include <slog2.h>

int slog2_register( slog2_buffer_set_config_t *config,
                   slog2_buffer_t *handles,
                   uint32_t flags );
```

Arguments:

config

A pointer to a slog2 buffer configuration object; see below.

handles

An array that the function fills with opaque handles for the buffers it allocates. The number of entries must be at least the value of the *num_buffers* member of the *config* structure.

flags

A bitwise OR of options for buffer management. No flags are currently defined; pass 0 for this argument.

Library:

libslog2

Use the -l slog2 option to gcc to link against this library.

Description:

The *slog2_register()* function register a new instance of a slog2 buffer set. Each buffer set contains one or more buffers; specify the size of each buffer as a multiple of 4 KB pages. The maximum number of buffers is specified by SLOG2_MAX_BUFFERS.

slog2_buffer_set_config_t and slog2_buffer_config_t structures

The slog2_buffer_config_t structure contains the configuration data for a single slog2 buffer:

```
typedef struct
{
    const char    *buffer_name;
```



```

    int          num_pages;
} slog2_buffer_config_t;

```

The fields include:

buffer_name

The name you want to use for the buffer.

num_pages

The number of 4 KB pages this buffer contains.

The `slog2_buffer_set_config_t` structure contains the configuration data for a `slog2` buffer set. Each buffer set contains one or more buffers; the maximum number of buffers is specified by `SLOG2_MAX_BUFFERS`.

```

typedef struct
{
    int          num_buffers;
    const char   *buffer_set_name;
    _Uint8t      verbosity_level;
    slog2_buffer_config_t buffer_config[ SLOG2_MAX_BUFFERS ];
} slog2_buffer_set_config_t;

```

num_buffers

The number of buffers to configure.

buffer_set_name

The process name, or other descriptor. We recommend that you use the name of your process for this.

verbosity_level

The minimum severity to log; one of:

- `SLOG2_SHUTDOWN` — shut down the system now (e.g., for OEM use).
- `SLOG2_CRITICAL` — unexpected unrecoverable error (e.g., hard disk error).
- `SLOG2_ERROR` — unexpected recoverable error (e.g., you need to reset a hardware controller).
- `SLOG2_WARNING` — expected error (e.g., parity error on a serial port).
- `SLOG2_NOTICE` — warning (e.g., out of paper).
- `SLOG2_INFO` — information (e.g., printing page 3).
- `SLOG2_DEBUG1` — debug messages (e.g., normal detail).
- `SLOG2_DEBUG2` — debug messages (e.g., fine detail).

All buffers in the set have the same level of verbosity.

buffer_config

An array of `slog2_buffer_config_t` structures that define the buffers.

Returns:

0

Success.

-1

An error occurred. The handle data isn't guaranteed.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <slog2.h>

/* Pull in the executable name.  argv[0] causes problems, so don't use it! */
extern char *__progname;

int main(int argc, char **argv)
{
    slog2_buffer_set_config_t    buffer_config;
    slog2_buffer_t              buffer_handle[2];

    /* A local variable used to demonstrate the slog2fa() API call below. */
    int                         some_number = 5108;

    /* You should use the name of your process to name the buffer set. */
    buffer_config.buffer_set_name = __progname;

    /* These two buffers are configured below. */
    buffer_config.num_buffers = 2;

    /* Request an initial verbosity level.

       We recommend that all components use SLOG2_INFO as the default
       verbosity level, then use SLOG2_DEBUG1 and SLOG2_DEBUG2 for higher
       levels of verbosity for in-depth investigations performed by
       developers or testers.

       All buffers within the set share the same verbosity level.
    */
    buffer_config.verbosity_level = SLOG2_INFO;

    /* !! NOTE: Please use a maximum of 32KB for the total of all your
       individual buffers !!
    */

    /* Configure the first buffer, using 7 x 4KB pages.  This larger
       buffer will be used for very chatty logging.  Our goal is to have
       30-60 seconds of history at any given time, so we will want to
       log at a rate of around one log line with a string of 16 bytes
       long every 150 milliseconds.
    */
    buffer_config.buffer_config[0].buffer_name = "hi_rate_logging";
    buffer_config.buffer_config[0].num_pages = 7;

    /* Configure the second buffer, which we will use for high level
       info logging that is very infrequent, but we want a longer history
       (hours or maybe even over a day or two).  This buffer uses 1 x 4KB.
    */
    buffer_config.buffer_config[1].buffer_name = "lo_rate_logging";
    buffer_config.buffer_config[1].num_pages = 1;

    /* Register the buffer set. */
```

```

if( -1 == slog2_register( &buffer_config, buffer_handle, 0 ) ) {
    fprintf( stderr, "Error registering slogger2 buffer!\n" );
    return -1;
}

/* slog2f()
- is the SLOWEST (though most convenient) of the slog2 APIs
- CANNOT be used with interrupts off because va_args handling
  could use interrupts
*/

slog2f( buffer_handle[0], 0, SLOG2_INFO,
        "Writing a formatted string into the buffer: %s", argv[0] );

/* slog2c()
- is the FASTEST slog2 API
- is interrupt safe
*/

slog2c( buffer_handle[0], 0, SLOG2_INFO,
        "Writing a constant string into the buffer." );

/* slog2fa()
- provides a VERY FAST and INTERRUPT SAFE alternative to slog2f()
- uses special formatting macros to ensure that va_args does
  NOT trigger interrupt use
*/

slog2fa( buffer_handle[0], 0, SLOG2_WARNING, "string:%s, some_number:%d",
        SLOG2_FA_STRING( "Hello world" ),
        SLOG2_FA_SIGNED( some_number ),
        SLOG2_FA_END );

/* Write something to the 'lo rate' buffer (i.e., buffer 1). */
slog2c( buffer_handle[1], 0, SLOG2_NOTICE,
        "This string will be logged." );

/* The verbosity level will prevent this from being written to the
   slog2 buffer (severity > verbosity_level). */

slog2c( buffer_handle[0], 0, SLOG2_DEBUG1,
        "This string should not be logged." );

return 0;
}

```

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slog2_reset()

Reset the state of slogger2 for the process

Synopsis:

```
#include <slog2.h>

int slog2_reset( void );
```

Library:

libslog2
Use the -l slog2 option to qcc to link against this library.

Description:

The *slog2_reset()* function resets the state of slogger2 within the process. All resources used by slogger2 within the process are released, so any existing buffer handles become invalid.

Returns:

0 for success, or -1 if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slog2_set_default_buffer()

Set the default slog2 logging buffer

Synopsis:

```
#include <slog2.h>

slog2_buffer_t slog2_set_default_buffer( slog2_buffer_t __buffer
);
```

Arguments:

buffer

One of:

- the handle of a buffer, allocated by [slog2_register\(\)](#) (p. 3052), where you want the default logs to go
- NULL if you want to disable default logging
- -1 to get the default buffer handle

Library:

lib slog2

Use the -l slog2 option to qcc to link against this library.

Description:

The *slog2_set_default_buffer()* function sets the default logging buffer. If you call the slog2 logging APIs with a NULL buffer handle, the log is written to the default logging buffer set by this function.

If *buffer* is NULL, default logging is disabled. If *buffer* is -1, the default buffer handle is returned.

Returns:

The previous default buffer handle.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slog2_set_verbosity()

Set the verbosity level for the entire buffer set

Synopsis:

```
#include <slog2.h>

int slog2_set_verbosity( slog2_buffer_t buffer,
                        uint8_t verbosity );
```

Arguments:

buffer

A handle for a buffer in the buffer set whose verbosity you want to set. This must not be NULL (i.e., you can't set the verbosity of the default buffer in this way).

verbosity

The new verbosity setting; one of:

- SLOG2_SHUTDOWN
- SLOG2_CRITICAL
- SLOG2_ERROR
- SLOG2_WARNING
- SLOG2_NOTICE
- SLOG2_INFO
- SLOG2_DEBUG1
- SLOG2_DEBUG2

For more information, see the entry for [slog2_register\(\)](#) (p. 3052).

Library:

lib slog2

Use the -l slog2 option to gcc to link against this library.

Description:

The *slog2_set_verbosity()* routine adjusts the verbosity of the *entire* buffer set that the given buffer belongs to. This value overrides the value set in the PPS object and updates the PPS value.

Returns:

0 on success, or -1 if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slogb()

Send a message to the system logger

Synopsis:

```
#include <stdio.h>
#include <sys/slog.h>

int slogb( int code,
           int severity,
           void * data,
           int size );
```

Arguments:

opcode

A combination of a *major* and *minor* code.

severity

The severity of the log message; see “[Severity levels](#) (p. 3064),” in the documentation for *slogf()*.

data

A block of raw data.

size

The size of the raw data.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *slog*()* functions send log messages to the system logger, `slogger`. To send formatted messages, use *slogf()* (p. 3063). If you have programs that scan log files for

specified codes, you can use *slogb()* or *slogi()* (p. 3066) to send a block of structures or `int`'s, respectively.

Errors:

Any value from the Errors section in *MsgSend()* (p. 1783), as well as:

EACCES

Insufficient permission to write to the log file.

EINVAL

The size of the data buffer exceeds 255×4 bytes, or an odd number of bytes is being read.

ENOENT

Invalid log file or directory specified, or `slogger` isn't running.

Examples:

See *slogf()* (p. 3063).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slogf()

Send a message to the system logger

Synopsis:

```
#include <stdio.h>
#include <sys/slog.h>

int slogf( int opcode,
           int severity,
           const char * fmt,
           ... );
```

Arguments:

opcode

A combination of a *major* and *minor* code. Create the *opcode* using the `_SLOG_SETCODE(major, minor)` macro that's defined in `<sys/slog.h>`.

The *major* and *minor* codes are defined in `<sys/slogcodes.h>`.

severity

The severity of the log message; see “[Severity levels](#) (p. 3064),” below.

fmt

A standard [printf\(\)](#) (p. 2133) string followed by *printf()* arguments.

The formatting characters that you use in the message determine any additional arguments.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *slog*()* functions send log messages to the system logger, `slogger`. To send formatted messages, use *slogf()*. If you have programs that scan log files for specified

codes, you can use [slogb\(\)](#) (p. 3061) or [slogi\(\)](#) (p. 3066) to send a block of structures or `int`'s, respectively.

The [vslogf\(\)](#) (p. 3630) function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of `<stdarg.h>`.

Severity levels

There are eight levels of severity defined. The lowest severity is 7 and the highest is 0. The default is 7.

Manifest Name	Severity value	Description
<code>_SLOG_SHUTDOWN</code>	0	Shut down the system NOW (e.g. for OEM use)
<code>_SLOG_CRITICAL</code>	1	Unexpected unrecoverable error (e.g. hard disk error)
<code>_SLOG_ERROR</code>	2	Unexpected recoverable error (e.g. needed to reset a hardware controller)
<code>_SLOG_WARNING</code>	3	Expected error (e.g. parity error on a serial port)
<code>_SLOG_NOTICE</code>	4	Warnings (e.g. out of paper)
<code>_SLOG_INFO</code>	5	Information (e.g. printing page 3)
<code>_SLOG_DEBUG1</code>	6	Debug messages (normal detail)
<code>_SLOG_DEBUG2</code>	7	Debug messages (fine detail)

Returns:

The size of the message sent to `slogger`, or -1 if an error occurs.

Errors:

Any value from the Errors section in [MsgSend\(\)](#) (p. 1783), as well as:

EACCES

Insufficient permission to write to the log file.

EINVAL

The size of the data buffer exceeds 255×4 bytes, or an odd number of bytes is being read.

ENOENT

Invalid log file or directory specified, or s logger isn't running.

Examples:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/slog.h>
#include <sys/slogcodes.h>

int main() {
    int i;

    for(i = 0 ; ; i++) {
        switch(rand() % 3) {
            case 0:
                slogb( _SLOG_SETCODE(_SLOGC_TEST, 0),
                    _SLOG_DEBUG1, &i, sizeof(i));
                break;

            case 1:
                slogi( _SLOG_SETCODE(_SLOGC_TEST, 1),
                    _SLOG_CRITICAL, 1, i);
                break;

            case 2:
                slogf( _SLOG_SETCODE(_SLOGC_TEST, 2),
                    _SLOG_ERROR,
                    "This is number %d", i);
                break;
        }
        sleep(1);
    }
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

slogi()

Send a message to the system logger

Synopsis:

```
#include <stdio.h>
#include <sys/slog.h>

int slogi( int code,
           int severity,
           int nargs,
           ... );
```

Arguments:

opcode

A combination of a *major* and *minor* code.

severity

The severity of the log message; see “[Severity levels](#) (p. 3064),” in the documentation for *slogf()*.

nargs

The number of integers to send. A maximum of 32 integers is allowed.

The additional arguments are the integers that you want to write.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *slog*()* functions send log messages to the system logger, `slogger`. To send formatted messages, use *slogf()* (p. 3063). If you have programs that scan log files for specified codes, you can use *slogb()* (p. 3061) or *slogi()* to send a block of structures or `int`'s, respectively.

Errors:

Any value from the Errors section in [MsgSend\(\)](#) (p. 1783), as well as:

EACCES

Insufficient permission to write to the log file.

EINVAL

The size of the data buffer exceeded 32 integers.

ENOENT

Invalid log file or directory specified, or slogger isn't running.

Examples:

See [slogf\(\)](#) (p. 3063)

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

_smalloc()

Allocate memory in blocks

Synopsis:

```
#include <malloc.h>

void* _smalloc( size_t size );
```

Arguments:

size

The size of the block to allocate, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `_smalloc()` function allocates space for an object of *size* bytes. Nothing is allocated when the *size* argument has a value of zero.



This function allocates memory in blocks of `_amblksiz` bytes; `_amblksiz` is a global variable defined in `<stdlib.h>`.

You must use `\_sfree\(\)` (p. 2923) to deallocate the memory allocated by `_smalloc()`.

Returns:

A pointer to the start of the allocated memory, or `NULL` if there's insufficient memory available, or if the requested *size* is zero.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

snprintf()

Write formatted output to a character array, up to a given maximum number of characters

Synopsis:

```
#include <stdio.h>

int snprintf( char* buf,
              size_t count,
              const char* format,
              ... );
```

Arguments:

buf

A pointer to the buffer where you want to function to store the formatted string.

count

The maximum number of characters to store in the buffer, including a terminating null character.

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `snprintf()` function is similar to [fprintf\(\)](#) (p. 702), except that `snprintf()` places the generated output (up to the specified maximum number of characters) into the character array pointed to by *buf*, instead of writing it to a file. The `snprintf()` function is similar to [sprintf\(\)](#) (p. 3151), but with boundary checking. A null character is placed at the end of the generated character string.

Returns:

The number of characters that would have been written into the array, not counting the terminating null character, had *count* been large enough. It does this even if *count* is zero; in this case *buf* can be NULL.

If an error occurred, *snprintf()* returns a negative value and sets [errno](#) (p. 539).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

/* Create temporary file names using a counter */

char namebuf[13];
int TempCount = 0;

char *make_temp_name( void )
{
    snprintf( namebuf, 13, "ZZ%.6o.TMP",
              TempCount++ );
    return( namebuf );
}

int main( void )
{
    FILE *tf1, *tf2;

    tf1 = fopen( make_temp_name(), "w" );
    tf2 = fopen( make_temp_name(), "w" );
    fputs( "temp file 1", tf1 );
    fputs( "temp file 2", tf2 );
    fclose( tf1 );
    fclose( tf2 );

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call this function in a signal handler if the data isn't floating point.

Be careful if you're using *snprintf()* to build a string one piece at a time. For example, this code:

```
len += snprintf(&buf[len], RECSIZE - 1 - len, ...);
```

could have a problem if *snprintf()* truncates the string. Without a separate test to compare *len* with *RECSIZE*, this code doesn't protect against a buffer overflow. After the call that truncates the output, *len* is larger than *RECSIZE*, and *RECSIZE - 1 - len* is a very large (unsigned) number; the next call generates unlimited output somewhere beyond the buffer.

socketmark()

Determine whether a socket is at the out-of-band mark

Synopsis:

```
#include <sys/socket.h>

int socketmark( int s );
```

Arguments:

s

The file descriptor of the socket that you want to check, as returned by [socket\(\)](#) (p. 3075).

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *socketmark()* function determines whether the socket specified by *s* is at the out-of-band data mark. If the protocol for the socket supports out-of-band data by marking the stream with an out-of-band data mark, *socketmark()* returns 1 when all data preceding the mark has been read and the out-of-band data mark is the first element in the receive queue.

The *socketmark()* function doesn't remove the out-of-band data mark from the stream.

Using this function between receive operations lets an application determine which data comes before and after out-of-band data.

Returns:

0

The socket isn't at the out-of-band data mark.

1

The socket is at the out-of-band data mark.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid file descriptor *s*.

ENOTTY

The *s* argument isn't the file descriptor of a valid socket.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

socket()

Create an endpoint for communication

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket( int domain,
            int type,
            int protocol );
```

Arguments:

domain

The communications domain that you want to use. This selects the protocol family that should be used. These families are defined in `<sys/socket.h>`.

type

The type of socket you want to create. This determines the semantics of communication. Here are the currently defined types:

- **SOCK_STREAM** — provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.
- **SOCK_DGRAM** — supports datagrams, which are connectionless, unreliable messages of a fixed (typically small) maximum length.
- **SOCK_RAW** — provides access to internal network protocols and interfaces. Available only to the superuser, this type isn't described here.

For more information, see below.

protocol

The particular protocol that you want to use with the socket. Normally, only a single protocol exists to support a particular socket type within a given protocol family. But if many protocols exist, you must specify one. The protocol number you give is particular to the communication domain where communication is to take place (see `/etc/protocols` in the *Utilities Reference*).

Library:

libsocket

Use the `-l` socket option to `gcc` to link against this library.

Description:

The `socket()` function creates an endpoint for communication and returns a descriptor.

SOCK_STREAM sockets

SOCK_STREAM sockets are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. To create a connection to another socket, call `connect()` (p. 348) call.

Once the socket is connected, you can transfer data by using `read()` (p. 2502) and `write()` (p. 3737) or some variant of `send()` (p. 2829) and `recv()` (p. 2533). When a session has been completed, a `close()` (p. 336) may be performed. Out-of-band data may also be transmitted (as described in `send()`) and received (as described in `recv()`).

The communications protocols used to implement a SOCK_STREAM socket ensure that data isn't lost or duplicated. If a piece of data that the peer protocol has buffer space for can't be successfully transmitted within a reasonable length of time, the connection is considered broken and calls will indicate an error by returning -1 and setting `errno` to ETIMEDOUT.

SOCK_DGRAM and SOCK_RAW sockets

With SOCK_DGRAM and SOCK_RAW sockets, datagrams can be sent to correspondents named in `send()` (p. 2829) calls. Datagrams are generally received with `recvfrom()` (p. 2536), which returns the next datagram with its return address.

Using the `ioctl()` call

You can use the `ioctl()` (p. 1198) call to specify a process group to receive a SIGURG signal when the out-of-band data arrives. The call may also enable nonblocking I/O and asynchronous notification of I/O events via SIGIO.

Socket-level options

The operation of sockets is controlled by socket-level options. These options are defined in the file `<sys/socket.h>`. Use `setsockopt()` (p. 2909) and `getsockopt()` (p. 962) to set and get options.

Returns:

A descriptor referencing the socket, or -1 if an error occurs (`errno` (p. 539) is set).

Errors:**EACCES**

Permission to create a socket of the specified type and/or protocol is denied.

EAFNOSUPPORT

The specified address family isn't supported.

EINTR

The call was interrupted by a signal.

EMFILE

The per-process descriptor table is full.

ENFILE

The system file table is full.

ENOBUFS

Insufficient buffer space available. The socket can't be created until sufficient resources are freed.

ENOMEM

Not enough memory.

EPROTONOSUPPORT

The protocol type or the specified protocol isn't supported within this domain.

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

By default, *socket()* communicates with the TCP/IP stack managing the `/dev/socket` portion of the namespace. You can change this behavior by setting the **SOCK** environment variable. For an example, see “Running multiple instances of the TCP/IP stack” in the TCP/IP Networking chapter of the QNX Neutrino *User's Guide*.

socketpair()

Create a pair of connected sockets

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair( int domain,
                int type,
                int protocol,
                int * fd[2] );
```

Arguments:

domain

The communications domain where the sockets are to be created.

type

The type of sockets to create.

protocol

The protocol to use with the sockets. A protocol of 0 causes *socketpair()* to use an unspecified default protocol appropriate for the requested socket type.

fd[2]

The 2-digit integer array where the file descriptors of the created socket pair are to be held.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *socketpair()* call creates an unnamed pair of connected sockets in the specified *domain*, of the specified *type*, using the optionally specified *protocol* argument. The file descriptors are returned in the vector *fd* and are identical. Data that's written to one of the file descriptors can be read from the other.

Valid types are described in [socket\(\)](#) (p. 3075).

If the *protocol* argument is nonzero, it must be a protocol that's understood by the address family. No such protocols are defined at this time.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAFNOSUPPORT

The specified address family isn't supported on this machine.

EFAULT

The address *sv* doesn't specify a valid part of the process address space.

EMFILE

Too many descriptors are in use by this process.

EOPNOTSUPP

The specified protocol doesn't support creation of socket pairs.

EPROTONOSUPPORT

The specified protocol isn't supported on this machine.

Examples:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>

#define CHAR_BUFSIZE 50

int main(int argc, char **argv) {
    int fd[2], len;
    char message[CHAR_BUFSIZE];

    if(socketpair(AF_LOCAL, SOCK_STREAM, 0, fd) == -1) {
        return 1;
    }

    /* If you write to fd[0], you read from fd[1], and vice versa. */

    /* Print a message into one end of the socket. */
    snprintf(message, CHAR_BUFSIZE, "A message written to fd[0]");
    write(fd[0], message, strlen(message) + 1);

    /* Print a message into the other end of the socket. */
    snprintf(message, CHAR_BUFSIZE, "A message written to fd[1]");
    write(fd[1], message, strlen(message) + 1);
}
```

```
/* Read from the first socket the data written to the second. */
len = read(fd[0], message, CHAR_BUFSIZE-1);
message[len] = '\0';
printf("Read from fd[0]: %s \n", message);

/* Read from the second socket the data written to the first. */
len = read(fd[1], message, CHAR_BUFSIZE-1);
message[len] = '\0';
printf("Read from fd[1]: %s \n", message);

close(fd[0]);
close(fd[1]);

return 0;
}
```

This program produces the following output:

```
Read from fd[0]: A message written to fd[1]
Read from fd[1]: A message written to fd[0]
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

SOCKSinit()

Initialize a connection with a SOCKS server

Synopsis:

```
#include <sys/select.h>

int SOCKSinit( char * progrname );
```

Arguments:

progrname

The name that you want to associate with the your program. The default is SOCKSclient.

Library:

libsocks

Use the -l socks option to gcc to link against this library.

Description:

The *SOCKSinit()* function initializes some defaults for the SOCKS library and also sets the program name that appears in the syslog output.

You don't have to call this function before making a SOCKS library call (but if you don't, a generic "SOCKSclient" appears instead of the program name).

For more information about SOCKS and its libraries, see the appendix, [SOCKS — A Basic Firewall](#).

Returns:

0

Success.

1

An error occurred ([errno](#) (p. 539) is set).

Classification:

[SOCKS](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

sopen()

Open a file for shared access

Synopsis:

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <share.h>

int sopen( const char* filename,
           int oflag,
           int share,
           ... );
```

Arguments:

filename

The path name of the file that you want to open.

oflag

Flags that specify the status and access modes of the file. This argument is a combination of the following bits (defined in <fcntl.h>):

- **O_RDONLY** — permit the file to be only read.
- **O_WRONLY** — permit the file to be only written.
- **O_RDWR** — permit the file to be both read and written.
- **O_APPEND** — cause each record that's written to be written at the end of the file.
- **O_CREAT** — create the file if it doesn't exist. This bit has no effect if the file already exists.
- **O_TRUNC** — truncate the file to contain no data if the file exists; this bit has no effect if the file doesn't exist.
- **O_EXCL** — open the file for exclusive access. If the file exists and you also specify **O_CREAT**, the open fails (that is, use **O_EXCL** to ensure that the file doesn't already exist).

share

The shared access for the file. This is a combination of the following bits (defined in <share.h>):

- **SH_COMPAT** — set compatibility mode.

- SH_DENYRW — prevent read or write access to the file.
- SH_DENYWR — prevent write access to the file.
- SH_DENYRD — prevent read access to the file.
- SH_DENYNO — permit both read and write access to the file.

If you set O_CREAT in *oflag*, you must also specify the following argument:

mode

An object of type `mode_t` that specifies the access mode that you want to use for a newly created file. For more information, see “[Access permissions](#) (p. 3165)” in the documentation for `stat()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sopen()` function opens a file at the operating system level for shared access. The name of the file to be opened is given by *filename*.

The file is accessed according to the access mode specified by *oflag*. You must specify O_CREAT if the file doesn't exist.

The sharing mode of the file is given by the *share* argument. The optional argument is the file permissions to be used when O_CREAT flag is on in the *oflag* mode; you must provide this when the file is to be created.

The `sopen()` function applies the current file permission mask to the specified permissions (see [umask\(\)](#) (p. 3557)).

Note that

```
open( path, oflag, ... );
```

is the same as:

```
sopen( path, oflag, SH_DENYNO, ... );
```



The `sopen()` function ignores advisory locks that you may have set by calling [fcntl\(\)](#) (p. 623).

Returns:

A descriptor for the file, or -1 if an error occurs while opening the file ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *oflag* are denied, or the file doesn't exist and write permission is denied for the parent directory of the file to be created.

EBUSY

Sharing mode (*share*) was denied due to a conflicting open.

EEXIST

The `O_CREAT` and `O_EXCL` flags are set, and the named file exists.

EISDIR

The named file is a directory, and the *oflag* argument specifies write-only or read/write access.

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

No more descriptors available (too many open files).

ENOENT

Path or file not found.

ENOSYS

The *sopen()* function isn't implemented for the filesystem specified in *path*.

Examples:

```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <share.h>

int main( void )
{
    int filedес;

    /* open a file for output */
    /* replace existing file if it exists */
}
```

```
    filedес = sopen( "file",
        O_WRONLY | O_CREAT | O_TRUNC,
        SH_DENYWR,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    /* read a file which is assumed to exist */
    filedес = sopen( "file", O_RDONLY, SH_DENYWR );

    /* append to the end of an existing file */
    /* write a new file if file doesn't exist */

    filedес = sopen( "file",
        O_WRONLY | O_CREAT | O_APPEND,
        SH_DENYWR,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
    return EXIT_SUCCESS;
}
```

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

sopenfd()

Open for shared access a file associated with a given descriptor

Synopsis:

```
#include <unistd.h>

int sopenfd( int fd,
             int oflag,
             int sflag );
```

Arguments:

fd

A file descriptor associated with the file that you want to open.

oflag

How you want to open the file; a combination of the following bits:

- O_RDONLY — permit the file to be only read.
- O_WRONLY — permit the file to be only written.
- O_RDWR — permit the file to be both read and written.
- O_APPEND — cause each record that's written to be written at the end of the file.
- O_TRUNC — if the file exists, truncate it to contain no data. This flag has no effect if the file doesn't exist.

sflag

How you want the file to be shared; a combination of the following bits:

- SH_COMPAT — set compatibility mode.
- SH_DENYRW — prevent read or write access to the file.
- SH_DENYWR — prevent write access to the file.
- SH_DENYRD — prevent read access to the file.
- SH_DENYNO — permit both read and write access to the file.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sopenfd()* function opens for shared access the file associated with the file descriptor, *fd*. The access mode, *oflag*, must be equal to or more restrictive than the access mode of the source *fd*.

Note that:

```
openfd( fd, oflag );
```

is the same as:

```
sopenfd( fd, oflag, SH_DENYNO );
```

Returns:

The file descriptor, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**EBADF**

Invalid file descriptor *fd*.

EACCES

The access mode specified by *oflag* isn't equal to or more restrictive than the access mode of the source *fd*.

EBUSY

Sharing mode (*sflag*) was denied due to a conflicting open.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

spawn()

Create and execute a new child process

Synopsis:

```
#include <spawn.h>

pid_t spawn( const char * path,
             int fd_count,
             const int fd_map[ ],
             const struct inheritance * inherit,
             char * const argv[ ],
             char * const envp[ ] );
```

Arguments:

path

The full path name of the executable.

fd_count

The number of entries in the *fd_map* array. If *fd_count* is 0, *fd_map* is ignored, and the child process inherits all file descriptors (except for the ones modified with *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag).

fd_map

An array of file descriptors that you want the child process to inherit. If *fd_count* isn't 0, *fd_map* must contain at least *fd_count* file descriptors, up to a maximum of `OPEN_MAX`. The first three entries (if specified) become the child process's standard input, standard output, and standard error (*stdin*, *stdout*, and *stderr*).

If you set *fdmap[X]* to `SPAWN_FDCLOSED` instead of to a valid file descriptor, the file descriptor *X* is closed in the child process.

For more information, see “[Mapping file descriptors](#) (p. 3092),” below.

inherit

A structure, of type `struct inheritance`, that indicates what you want the child process to inherit from the parent. For more information, see “[inheritance structure](#) (p. 3093),” below.

argv

A pointer to an argument vector. The value in *argv[0]* can't be `NULL`, and should represent the filename of the program being loaded. The last member of *argv* must be a `NULL` pointer. The value of *argv* can't be `NULL`.

envp

A pointer to an array of character pointers, each pointing to a string defining an environment variable. The array is terminated with a `NULL` pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable. If the value of *envp* is `NULL`, then the child process inherits the environment of the parent process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *spawn()* function creates and executes a new child process, named in *path*.



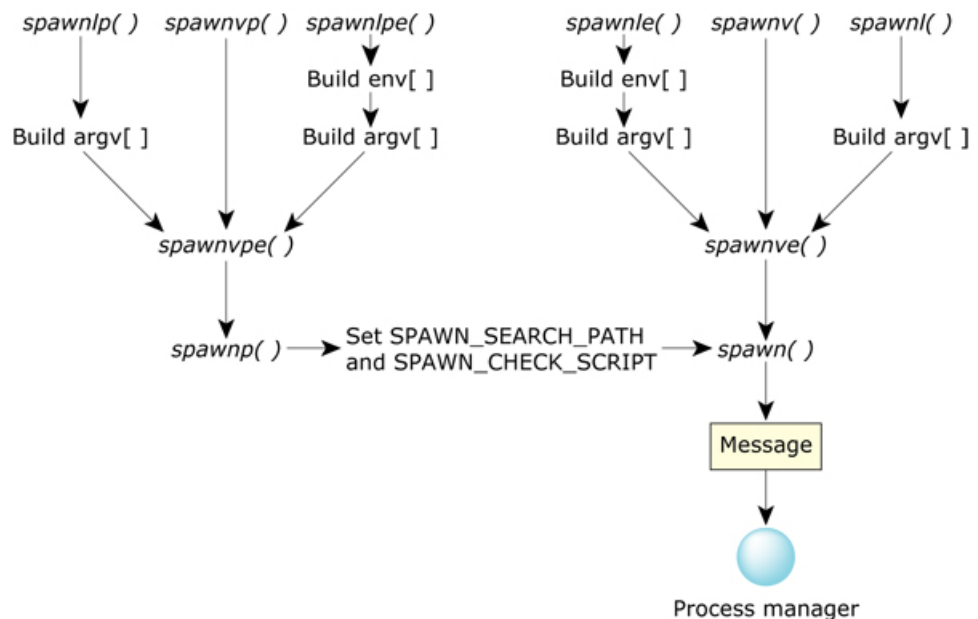
- In order to create a child process, your process must have the `PROCmgr_AID_SPAWN` ability enabled. In order to create a new application ID for the child process, your process must have the `PROCmgr_AID_CHILD_NEWAPP` ability enabled. For more information, see *procmgr_ability()* (p. 2141).
- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The *spawn()* function is a QNX Neutrino function (based on the POSIX 1003.1d *draft standard*); for greater portability, use *posix_spawn()* (p. 2037). The C library also includes several specialized *spawn*()* functions. Their names consist of *spawn* followed by several letters:

This suffix:	Indicates the function takes these arguments:
e	An array of environment variables.

This suffix:	Indicates the function takes these arguments:
<code>l</code>	A NULL-terminated list of arguments to the program.
<code>p</code>	A relative path. If the path doesn't contain a slash, the PATH environment variable is searched for the program. This suffix also lets the <code>#!</code> construction work; see <code>SPAWN_CHECK_SCRIPT</code> , below.
<code>v</code>	A vector of arguments to the program.

As shown below, these functions eventually call `spawn()`, which in turn sends a message to the process manager, `procnto`:



The child process inherits the following attributes of the parent process:

- process group ID (unless `SPAWN_SETGROUP` is set in `inherit.flags`)
- session membership
- real user ID and real group ID
- effective user ID and effective group ID
- supplementary group IDs
- priority and scheduling policy
- current working directory and root directory
- file-creation mask
- signal mask (unless `SPAWN_SETSIGMASK` is set in `inherit.flags`)
- signal actions specified as `SIG_DFL`

- signal actions specified as `SIG_IGN` (except the ones modified by *inherit.sigdefault* when `SPAWN_SETSIGDEF` is set in *inherit.flags*).

The child process has several differences from the parent process:

- Signals set to be caught by the parent process are set to the default action (`SIG_DFL`).
- The child process's *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are tracked separately from the parent's.
- File locks set by the parent aren't inherited.
- Per-process timers created by the parent aren't inherited.
- Memory locks and mappings set by the parent aren't inherited.
- The child doesn't inherit the parent's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

The child process also has these differences from the parent process if you haven't set the `SPAWN_EXEC` flag:

- The number of seconds left until a `SIGALRM` signal would be generated is set to zero for the child process.
- The set of pending signals for the child process is empty.

If the child process is spawned on a remote node, the process group ID and the session membership aren't set; the child process is put into a new session and a new process group.

The child process can access its environment by using the [environ](#) (p. 528) global variable (found in `<unistd.h>`).

If the *path* is on a filesystem mounted with the `ST_NOSUID` flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the child process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the child process is set to the owner ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the child process is set to the group ID of *path*.

The real user ID, real group ID and supplementary group IDs of the child process remain the same as those of the parent process. The effective user ID and effective group ID of the child process are saved as the saved set-user ID and the saved set-group ID used by the [setuid\(\)](#) (p. 2917).



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Mapping file descriptors

As described above, you can use the *fd_count* and *fd_map* arguments to specify which file descriptors you want the child process to inherit.

The number used for a file descriptor in the child process depends on its position in *fd_map*, not on the actual file descriptor in the parent. For example, suppose the parent has file descriptors valued 1, 3, and 5 that you want to have in the child process. If you specify:

```
int fd_map = { 1, 3, 5 };
```

then the mapping is as follows:

In the child, this fd:	Is the same as this fd in the parent:
0	1
1	3
2	5

If 1, 3, and 5 are the only file descriptors open in the parent, and you specify an *fd_count* of 0, the function duplicates all the file descriptors:

In the child, this fd:	Is the same as this fd in the parent:
1	1
3	3
5	5

and file descriptors 0, 2, 4, and 6 (and higher) are closed.

If you're using an explicit *fd_map*, then in order to get numerically matching file descriptors in the child and the parent, you must fill the holes in the map with `SPAWN_FDCLOSE`:

```
int fd_map = { SPAWN_FDCLOSE, 1, SPAWN_FDCLOSE, 3, SPAWN_FDCLOSE, 5 };
```



Be careful if you use `SPAWN_FDCLOSE` for file descriptors 0, 1, and 2. If the child process calls `open()` (p. 1905), the function assigns the lowest numbered unused file descriptor. If, for example, you set `stdout` or 1 to `SPAWN_FDCLOSED`, the next `open()` in the spawned application is assigned file descriptor 1. In this case, `printf()` (p. 2133) would write to whatever resource was opened, which may or may not be what you intended.

It's safer to set 0, 1 and 2 to be the file descriptor from an `open()` of a known resource (such as `/dev/null`) in the parent, and then pass those file descriptors to the child.

inheritance structure

The `inheritance` structure contains at least these members:

unsigned long flags

Zero or more of the following bits:

- **SPAWN_ALIGN_DEFAULT** — use the system's default settings for alignment.
- **SPAWN_ALIGN_FAULT** — try to always fault data misalignment references.
- **SPAWN_ALIGN_NOFAULT** — don't fault on misalignment; attempt to fix it (this may be slow).
- **SPAWN_CHECK_SCRIPT** — start a shell, passing *path* as a script.
- **SPAWN_DEBUG** — debug process (this is used only for debugging the kernel itself).
- **SPAWN_EXEC** — cause the spawn to act like *exec*()*: replace the calling program in memory with the newly loaded program. If successful, no return is made to the calling program.
- **SPAWN_EXPLICIT_CPU** — (QNX Neutrino Core OS 6.3.2 or later) Set the runmask and inherit mask equal to the *runmask* member of the inheritance structure. If this flag isn't set, the child inherits the inherit mask of the calling thread.
- **SPAWN_EXPLICIT_SCHED** — set the scheduling policy to the value of the *policy* member, and the scheduling parameters to the value of the *param* member.
- **SPAWN_HOLD** — hold a process for debugging (i.e. send the SIGSTOP signal to the process before it executes its first instruction).
- **SPAWN_NEWAPP** — assign a new application ID to the process. If you don't set this flag, the process inherits the parent's application ID.
- **SPAWN_NOZOMBIE** — prevent the child process from becoming a zombie on its death. No child return or exit information will be available.
- **SPAWN_PADDR64_SAFE** — indicate that the process is known to operate safely with 64-bit addressing or doesn't care about the physical memory location.
- **SPAWN_SEARCH_PATH** — search the **PATH** environment variable for the executable.
- **SPAWN_SETGROUP** — set the child's process group to the value in the *pgroup* member. If this flag isn't set, the child process is part of the current process group.
- **SPAWN_SETND** — spawn the child process on the node specified by the *nd* member.
- **SPAWN_SETSID** — make the new process a session leader.

- **SPAWN_SETSIGDEF** — use the *sigdefault* member to specify the child process's set of defaulted signals. If you don't specify this flag, the child process inherits the parent process's signal actions.
- **SPAWN_SETSIGIGN** — set the handling for signals defined in the *sigignore* member to **SIG_IGN**.
- **SPAWN_SETSIGMASK** — use the *sigmask* member to specify the child process's signal mask.
- **SPAWN_SETSTACKMAX** — set the maximum stack size to the value of the *stack_max* member.
- **SPAWN_TCSETPGROUP** — start a new terminal group.

The `<spawn.h>` file also defines **SPAWN_ALIGN_MASK**. It's a mask for the alignment flags listed above.

pid_t pgroup

The child process's group if you specify **SPAWN_SETGROUP** in the *flags* member.

If **SPAWN_SETGROUP** is set in *inherit.flags* and *inherit.pgroup* is set to **SPAWN_NEWPGROUP**, the child process starts a new process group with the process group ID set to its process ID.

int runmask

(QNX Neutrino Core OS 6.3.2 or later) If you specify **SPAWN_EXPLICIT_CPU** in the *flags* member, the child process's runmask and inherit mask are set to the value of this member. For more information, see the *Multicore Processing User's Guide*.

sigset_t sigmask

The child process's signal mask if you specify **SPAWN_SETSIGMASK** in the *flags* member.

sigset_t sigdefault

The child process's set of defaulted signals if you specify **SPAWN_SETSIGDEF** in the *flags* member.

sigset_t sigignore

The child process's set of ignored signals if you specify **SPAWN_SETSIGIGN** in the *flags* member.

unsigned long stack_max

The maximum stack size for the child process, if you set `SPAWN_SETSTACKMAX` in the *flags* member.

int *policy*

The scheduling policy for the child process, if you set `SPAWN_EXPLICIT_SCHED` in the *flags* member. The policy must be one of the following:

- `SCHED_FIFO` — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
- `SCHED_RR` — similar to `SCHED_FIFO`, except that threads at the same priority level timeslice (round robin) every 4 × the clock period (see [ClockPeriod\(\)](#) (p. 329)).
- `SCHED_OTHER` — currently the same as `SCHED_RR`.
- `SCHED_SPORADIC` — sporadic scheduling.

uint32_t *nd*

The node descriptor of the remote node on which to spawn the child process. This member is used only if you set `SPAWN_SETND` in the *flags* member.



If you want to *spawn()* remotely, set the *nd* member to the node descriptor. See the [netmgr_strtnd\(\)](#) (p. 1883) function.

struct `sched_param` *param*

Scheduling parameters for the child process, if you set `SPAWN_EXPLICIT_SCHED` in the *flags* member. For more information, see the documentation for [sched_param](#) (p. 2731).

Returns:

The process ID of the child process, or -1 if an error occurs ([errno](#) (p. 539) is set).



If you set `SPAWN_EXEC` in the *flags* member of the *inheritence* structure, *spawn()* doesn't return, unless an error occurred.

Errors:

E2BIG

The number of bytes used by the argument list and environment list of the new child process is greater than `ARG_MAX` bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the child process's file doesn't have the execute bit set or *path*'s filesystem was mounted with the ST_NOEXEC flag.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An entry in *fd_map* refers to an invalid file descriptor, or an error occurred duplicating open file descriptors to the new process.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *argv[0]* is NULL).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *path* exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *path* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable. (This error doesn't occur if SPAWN_CHECK_SCRIPT is set in the *flags* member of the *inheritance* structure.)

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawn()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

spawnl()

Spawn a child process, given a list of arguments

Synopsis:

```
#include <process.h>

int spawnl( int mode,
            const char * path,
            const char * arg0,
            const char * arg1...,
            const char * argn,
            NULL );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

path

The full path name of the executable.

arg0, ... argn, NULL

The arguments that you want to pass to the new process. You must pass at least *arg0*, which by convention is a pointer to the name of the new child process, and can't be NULL. You must terminate the list with an argument of NULL.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnl()` function creates and executes a new child process, named in *path* with a NULL-terminated list of arguments in *arg0* ... *argn*. This function calls [spawnve\(\)](#) (p. 3134).

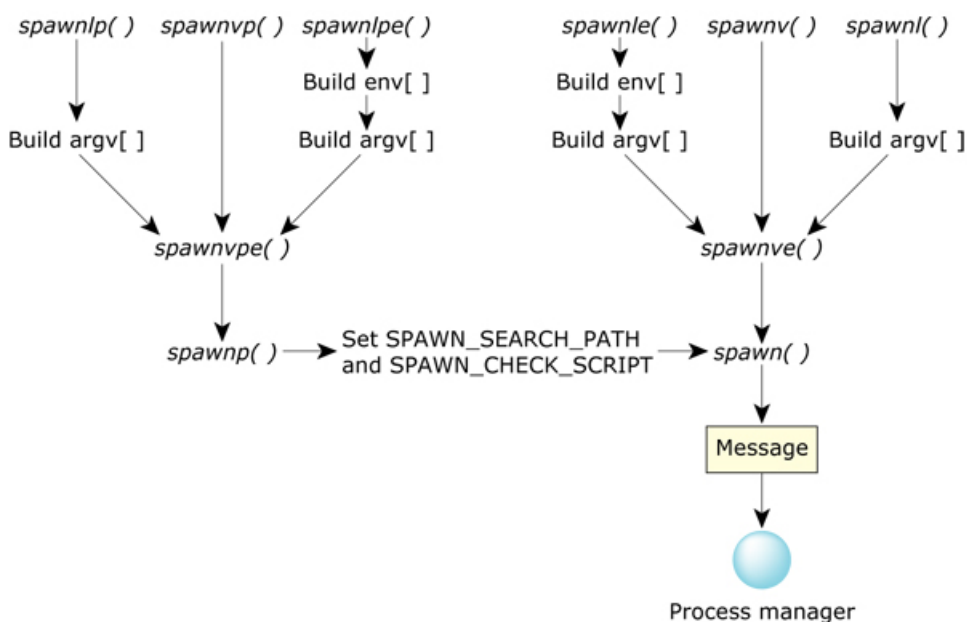
- In order to create a child process, your process must have the `PROC_MGR_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnl()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It builds an `argv[ ]` array before calling `spawn()`. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process.

The child process inherits the parent's environment. The environment is the collection of environment variables whose values that have been defined with the `export` shell command, the `env` utility, or by the successful execution of the [putenv\(\)](#) (p. 2454) or [setenv\(\)](#) (p. 2853) function. A program may read these values with the [getenv\(\)](#) (p. 838) function.



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The `spawnl()` function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see “ Status macros (p. 3652)” in the documentation for wait() .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote node. You can't get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned ([errno](#) (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is `P_WAIT`, and the spawned process terminated before the call to *waitpid()* was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is `NULL`, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *path* exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.

ENOENT

The file identified by the *path* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnl()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Examples:

Run myprog as if the user had typed:

```
myprog ARG1 ARG2
```

at the command-line:

```
#include <stddef.h>
#include <process.h>

int exit_val;
...
exit_val = spawnl( P_WAIT, "myprog",
                  "myprog", "ARG1", "ARG2", NULL );
...
```

The program is found if myprog is in the current working directory.

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is P_WAIT, this function is a cancellation point.

spawnle()

Spawn a child process, given a list of arguments and an environment

Synopsis:

```
#include <process.h>

int spawnle( int mode,
             const char * path,
             const char * arg0,
             const char * arg1...,
             const char * argn,
             NULL,
             const char * envp[] );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

path

The full path name of the executable.

arg0, ... argn, NULL

The arguments that you want to pass to the new process. You must pass at least *arg0*, which by convention is a pointer to the name of the new child process, and can't be `NULL`. You must terminate the list with an argument of `NULL`.

envp

NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnle()` function creates and executes a new child process, named in *path* with NULL-terminated list of arguments in *arg0* ... *argn* and with the environment specified in *envp*. This function calls [spawnve\(\)](#) (p. 3134).

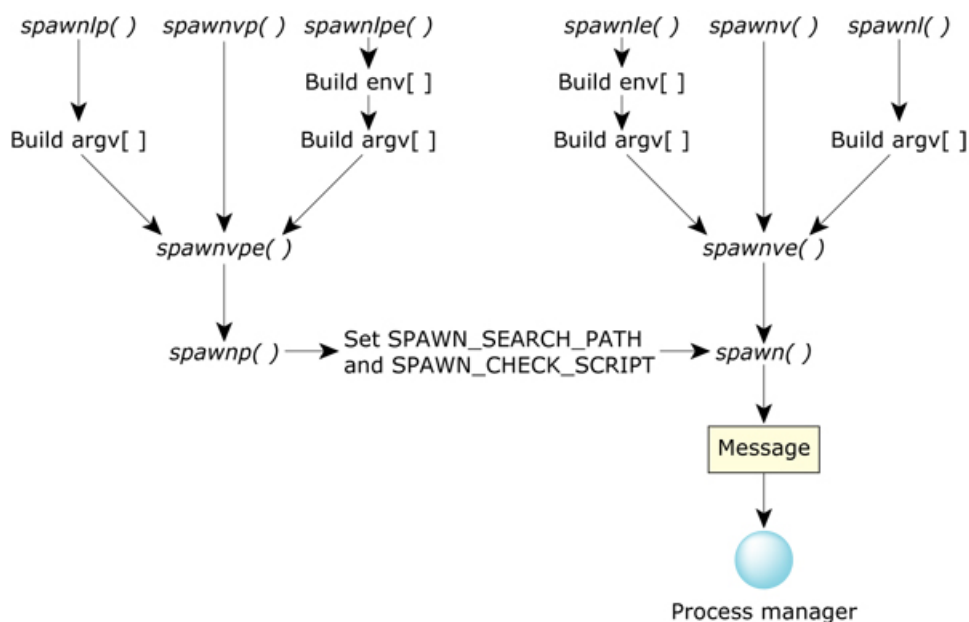
- In order to create a child process, your process must have the PROCmgr_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnle()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It builds `argv[ ]` and `envp[ ]` arrays before calling `spawn()`. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process.

If *envp* is NULL, the child process inherits the environment of the parent process. The new process can access its environment by using the [environ](#) (p. 528) global variable (found in `<unistd.h>`).



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The *spawnle()* function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see “ Status macros (p. 3652)” in the documentation for <i>wait()</i> .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote

<i>mode</i>	Return value
	node. You can't get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned (*errno* (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list or environment list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is P_WAIT, and the spawned process terminated before the call to *waitpid()* was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is NULL, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *path* exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *path* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnle()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Examples:

Run myprog as if the user had typed:

```
myprog ARG1 ARG2
```

at the command-line:

```
#include <stddef.h>
#include <process.h>

char *env_list[] = { "SOURCE=MYDATA",
                    "TARGET=OUTPUT",
                    "lines=65",
                    NULL
};
```

```
spawnle( P_WAIT, "myprog",  
         "myprog", "ARG1", "ARG2", NULL,  
         env_list );
```

The program is found if `myprog` is in the current working directory. The environment for the child program consists of the three environment variables ***SOURCE***, ***TARGET*** and ***lines***.

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is `P_WAIT`, this function is a cancellation point.

spawnlp()

Spawn a child process, given a list of arguments and a relative path

Synopsis:

```
#include <process.h>

int spawnlp( int mode,
             const char * file,
             const char * arg0,
             const char * arg1...,
             const char * argn,
             NULL );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

file

The name of the executable file. If this argument contains a slash, it's used as the pathname of the executable; otherwise, the function searches for *file* in the directories listed in the **PATH** environment variable.

arg0, ... argn, NULL

The arguments that you want to pass to the new process. You must pass at least *arg0*, which by convention is a pointer to the name of the new child process, and can't be `NULL`. You must terminate the list with an argument of `NULL`.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnlp()` function creates and executes a new child process, named in *file* with NULL-terminated list of arguments in *arg0* ... *argn*. This function calls [spawnvpe\(\)](#) (p. 3145).

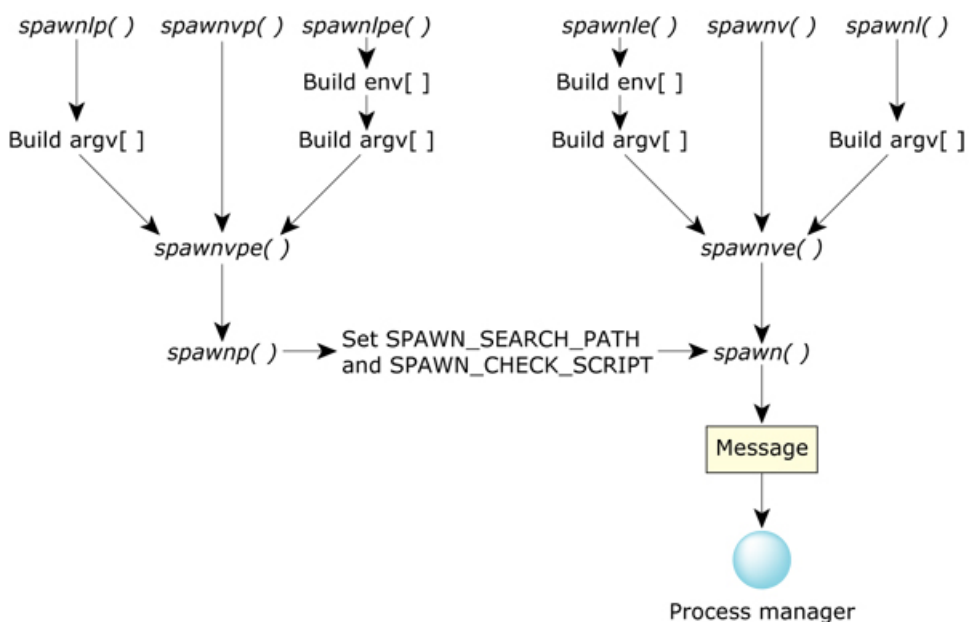
- In order to create a child process, your process must have the `PROC_MGR_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnlp()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It builds an `argv[ ]` array before calling `spawnp()`. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process.

The child process inherits the parent's environment. The environment is the collection of environment variables whose values that have been defined with the `export` shell command, the `env` utility, or by the successful execution of the [putenv\(\)](#) (p. 2454) or [setenv\(\)](#) (p. 2853) function. A program may read these values with the [getenv\(\)](#) (p. 838) function.



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The `spawnlp()` function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see “ Status macros (p. 3652)” in the documentation for wait() .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote node. You can't get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned ([errno](#) (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is `P_WAIT`, and the spawned process terminated before the call to *waitpid()* was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is `NULL`, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *file* exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.

ENOENT

The file identified by the *file* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnlp()* function isn't implemented for the filesystem specified in *file*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is P_WAIT, this function is a cancellation point.

spawnlpe()

Spawn a child process, given a list of arguments, an environment, and a relative path

Synopsis:

```
#include <process.h>

int spawnlpe( int mode,
               const char * file,
               const char * arg0,
               const char * arg1...,
               const char * argn,
               NULL,
               const char * envp[] );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

file

The name of the executable file. If this argument contains a slash, it's used as the pathname of the executable; otherwise, the function searches for *file* in the directories listed in the **PATH** environment variable.

arg0, ... argn, NULL

The arguments that you want to pass to the new process. You must pass at least *arg0*, which by convention is a pointer to the name of the new child process, and can't be NULL. You must terminate the list with an argument of NULL.

envp

NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable.

Library:**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnlpe()` function creates and executes a new child process, named in *file* with NULL-terminated list of arguments in *arg0* ... *argn* and with the environment specified in *envp*. This function calls [spawnvpe\(\)](#) (p. 3145).

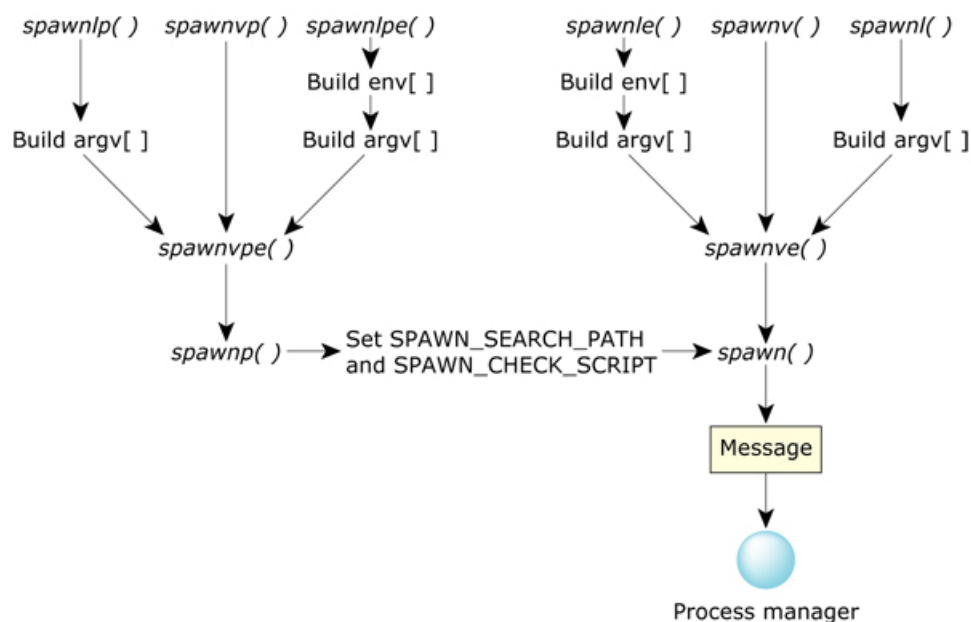
- In order to create a child process, your process must have the PROCmgr_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnlpe()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It builds `argv[ ]` and `envp[ ]` arrays before calling `spawnp()`. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process.

If the value of *envp* is NULL, then the child process inherits the environment of the parent process. The new process can access its environment by using the [environ](#) (p. 528) global variable (found in <unistd.h>).



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The *spawnlpe()* function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see “ Status macros (p. 3652)” in the documentation for <i>wait()</i> .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote

<i>mode</i>	Return value
	node. You can't get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned (*errno* (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list or environment list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is P_WAIT, and the spawned process terminated before the call to *waitpid()* was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is NULL, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *file* exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *file* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnlpe()* function isn't implemented for the filesystem specified in *file*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is P_WAIT, this function is a cancellation point.

spawnp()

Create and execute a new child process, given a relative path

Synopsis:

```
#include <spawn.h>

pid_t spawnp( const char * file,
               int fd_count,
               const int fd_map[ ],
               const struct inheritance * inherit,
               char * const argv[ ],
               char * const envp[ ] );
```

Arguments:

file

If this argument contains a slash, it's used as the pathname of the executable; otherwise, the **PATH** environment variable is searched for *file*.

fd_count

The number of entries in the *fd_map* array. If *fd_count* is 0, *fd_map* is ignored, and the child process inherits all file descriptors (except for the ones modified with *fcntl()*'s [FD_CLOEXEC](#) (p. ?) flag).

fd_map

An array of file descriptors that you want the child process to inherit. If *fd_count* isn't 0, *fd_map* must contain at least *fd_count* file descriptors, up to a maximum of **OPEN_MAX**. The first three entries (if specified) become the child process's standard input, standard output, and standard error (*stdin*, *stdout*, and *stderr*).

If you set *fdmap[X]* to **SPAWN_FDCLOSED** instead of to a valid file descriptor, the file descriptor *X* is closed in the child process.

For more information, see “[Mapping file descriptors](#) (p. 3092)” in the documentation for *spawn()*.

inherit

A structure, of type `struct inheritance`, that indicates what you want the child process to inherit from the parent. For more information, see “[inheritance structure](#) (p. 3093),” in the documentation for *spawn()*.



If you want to *spawnp()* remotely, set the *nd* member of the inheritance structure to the node descriptor. See the [netmgr_strtond\(\)](#) (p. 1883) function.

argv

A pointer to an argument vector. The value in *argv[0]* can't be NULL, and should represent the filename of the program being loaded. The last member of *argv* must be a NULL pointer. The value of *argv* can't be NULL.

envp

A pointer to an array of character pointers, each pointing to a string defining an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable. If the value of *envp* is NULL, then the child process inherits the environment of the parent process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *spawnp()* function creates and executes a new child process, named in *file*. It sets the SPAWN_CHECK_SCRIPT and SPAWN_SEARCH_PATH flags, and then calls [spawn\(\)](#) (p. 3089).

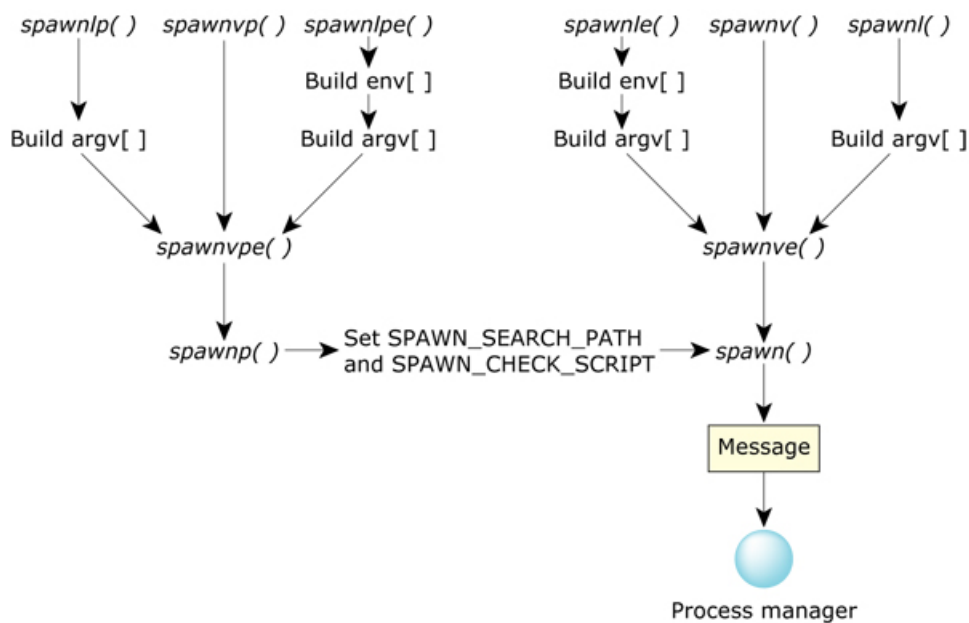


- In order to create a child process, your process must have the PROC_MGR_AID_SPAWN ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The *spawnp()* function is a QNX Neutrino function (based on the POSIX 1003.1d *draft standard*); for greater portability, use *posix_spawn()* (p. 2037). The C library also includes several specialized *spawn*()* functions. Their names consist of *spawn* followed by several letters:

This suffix:	Indicates the function takes these arguments:
e	An array of environment variables.
l	A NULL-terminated list of arguments to the program.
p	A relative path. If the path doesn't contain a slash, the PATH environment variable is searched for the program. This suffix also lets the #! construction work; see SPAWN_CHECK_SCRIPT in the documentation for <i>spawn()</i> .
v	A vector of arguments to the program.

Most of the *spawn*()* functions do a lot of work before a message is sent to *procnto*:



The child process inherits the following attributes of the parent process:

- process group ID (unless SPAWN_SETGROUP is set in *inherit.flags*)
- session membership
- real user ID and real group ID
- effective user ID and effective group ID
- supplementary group IDs

- priority and scheduling policy
- current working directory and root directory
- file-creation mask
- signal mask (unless SPAWN_SETSIGMASK is set in *inherit.flags*)
- signal actions specified as SIG_DFL
- signal actions specified as SIG_IGN (except the ones modified by *inherit.sigdefault* when SPAWN_SETSIGDEF is set in *inherit.flags*).

The child process has several differences from the parent process:

- Signals set to be caught by the parent process are set to the default action (SIG_DFL).
- The child process's *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are tracked separately from the parent's.
- The number of seconds left until a SIGALRM signal would be generated is set to zero for the child process.
- The set of pending signals for the child process is empty.
- File locks set by the parent aren't inherited.
- Per-process timers created by the parent aren't inherited.
- Memory locks and mappings set by the parent aren't inherited.
- The child doesn't inherit the parent's default timer tolerance (set by a call to [procmgr_timer_tolerance\(\)](#) (p. 2178)).

If the child process is spawned on a remote node, the process group ID and the session membership aren't set; the child process is put into a new session and a new process group.

The child process can access its environment by using the [environ](#) (p. 528) global variable (found in <unistd.h>).

If the *file* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the child process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the child process is set to the owner ID of *file*. Similarly, if the set-group ID mode bit is set, the effective group ID of the child process is set to the group ID of *file*. The real user ID, real group ID and supplementary group IDs of the child process remain the same as those of the parent process. The effective user ID and effective group ID of the child process are saved as the saved set-user ID and the saved set-group ID used by the [setuid\(\)](#) (p. 2917).

Suppose you call *spawnp()* like this:

```
spawnp(file, fd_count, fd_map, inherit, argv, envp);
```

The *spawnp()* function does the following:

- creates an empty attribute if *inherit* is NULL

- sets the `CHECK_SCRIPT` flag
- sets the `SPAWN_SEARCH_PATH` flag

and then calls `spawn()`.



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The process ID of the child process, or -1 if an error occurs ([errno](#) (p. 539) is set).



If you set `SPAWN_EXEC` in the `flags` member of the `inheritance` structure, `spawnp()` doesn't return, unless an error occurred.

Errors:

E2BIG

The number of bytes used by the argument list and environment list of the new child process is greater than `ARG_MAX` bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set or *file*'s filesystem was mounted with the `ST_NOEXEC` flag.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An entry in *fd_map* refers to an invalid file descriptor, or an error occurred duplicating open file descriptors to the new process.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *argv[0]* is `NULL`).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *file* exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *file* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnp()* function isn't implemented for the filesystem specified in *file*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

spawnv()

Spawn a child process, given a vector of arguments

Synopsis:

```
#include <process.h>

int spawnv( int mode,
            const char * path,
            char * const argv[] );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

path

The full path name of the executable.

argv

A pointer to an argument vector; this argument can't be NULL. The value in `argv[0]` can't be NULL, and should represent the filename of the program being loaded. The last member of *argv* *must* be a NULL pointer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnv()` function creates and executes a new child process, named in *path* with the NULL-terminated list of arguments in the *argv* vector. This function calls [spawnvpe\(\)](#) (p. 3145).

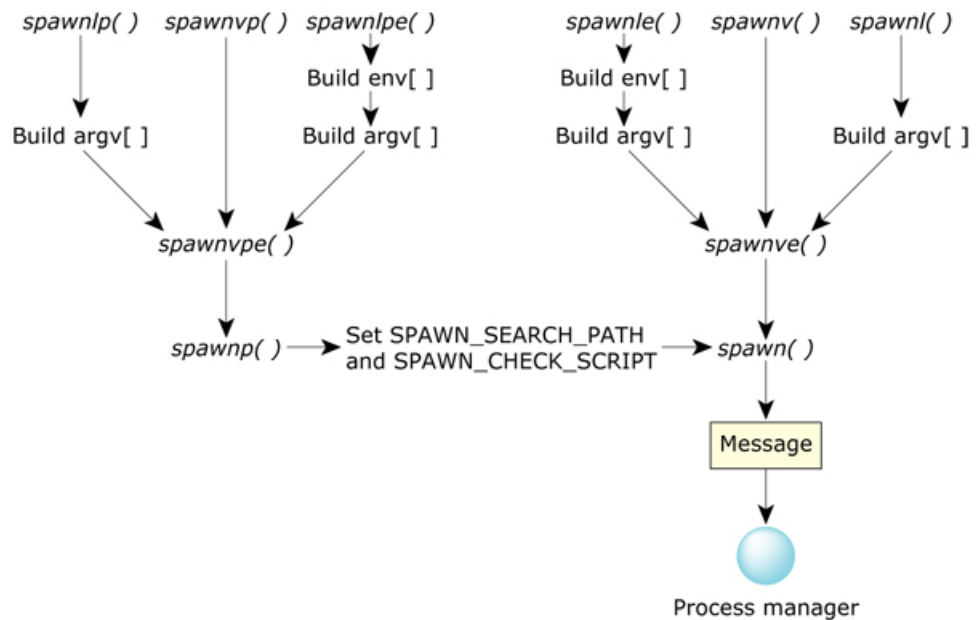
- In order to create a child process, your process must have the `PROCmgr_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnv()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It calls `spawnvpe()` before calling `spawn()`. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



The child process inherits the parent's environment. The environment is the collection of environment variables whose values that have been defined with the `export` shell command, the `env` utility, or by the successful execution of the [putenv\(\)](#) (p. 2454) or [setenv\(\)](#) (p. 2853) function. A program may read these values with the [getenv\(\)](#) (p. 838) function.



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The *spawnv()* function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see “ Status macros (p. 3652)” in the documentation for <i>wait()</i> .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the <i>waitpid()</i> (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote node. You can't get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned (*errno* (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is P_WAIT, and the spawned process terminated before the call to *waitpid()* was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is NULL, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *path* exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *path* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnv()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed

because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Examples:

Run myprog as if a user had typed:

```
myprog ARG1 ARG2
```

at the command-line:

```
#include <stddef.h>
#include <process.h>

char *arg_list[] = { "myprog", "ARG1", "ARG2", NULL };
...
spawnv( P_WAIT, "myprog", arg_list );
```

The program is found if myprog is in the current working directory.

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is P_WAIT, this function is a cancellation point.

spawnve()

Spawn a child process, given a vector of arguments and an environment

Synopsis:

```
#include <process.h>

int spawnve( int mode,
             const char * path,
             char * const argv[],
             char * const envp[] );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

path

The full path name of the executable.

argv

A pointer to an argument vector; this argument can't be **NULL**. The value in `argv[0]` can't be **NULL**, and should represent the filename of the program being loaded. The last member of `argv` *must* be a **NULL** pointer.

envp

NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *spawnve()* function creates and executes a new child process, named in *path* with the NULL-terminated list of arguments in the *argv* vector.

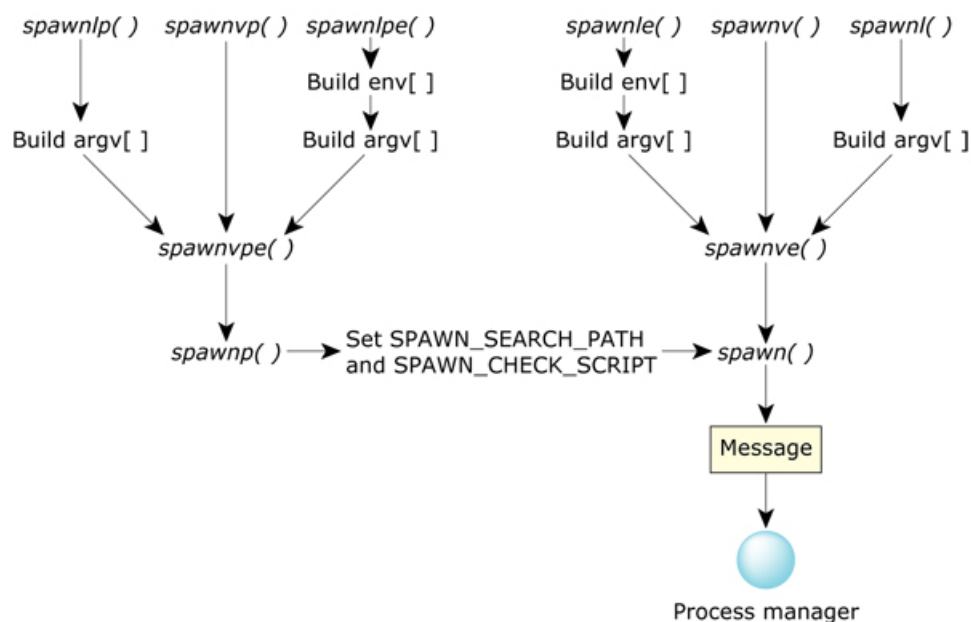
- In order to create a child process, your process must have the `PROC_MGR_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The *spawnve()* function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It calls *spawn()*. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the *spawn*()* functions do a lot of work before a message is sent to `procnto`:



If the value of *envp* is NULL, then the child process inherits the environment of the parent process. The new process can access its environment by using the [environ](#) (p. 528) global variable (found in <unistd.h>).

The following arguments are passed to the underlying call (*spawn()*):

spawnve(mode, path, argv, envp)

- Create temporary attribute structure and set the flags member based on the *mode*:
 - 0 if *mode* = P_WAIT/P_NOWAIT
 - SPAWN_EXEC if *mode* = P_OVERLAY
 - SPAWN_NOZOMBIE if *mode* = P_NOWAITO
- Call *spawn*(path, 0, 0, attr, argv, envp).



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The *spawnve()* function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see " Status "

<i>mode</i>	Return value
	macros (p. 3652)” in the documentation for wait() .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote node. You can't get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned ([errno](#) (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list or environment list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is P_WAIT, and the spawned process terminated before the call to [waitpid\(\)](#) was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is NULL, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *path* exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *path* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process's file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnve()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is P_WAIT, this function is a cancellation point.

spawnvp()

Spawn a child process, given a vector of arguments and a relative path

Synopsis:

```
#include <process.h>

int spawnvp( int mode,
             const char * file,
             char * const argv[] );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

file

The name of the executable file. If this argument contains a slash, it's used as the pathname of the executable; otherwise, the function searches for *file* in the directories listed in the **PATH** environment variable.

argv

A pointer to an argument vector; this argument can't be NULL. The value in *argv[0]* can't be NULL, and should represent the filename of the program being loaded. The last member of *argv* *must* be a NULL pointer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnvp()` function creates and executes a new child process, named in *file* with the NULL-terminated list of arguments in the *argv* vector. This function calls `spawnvpe()` (p. 3145).

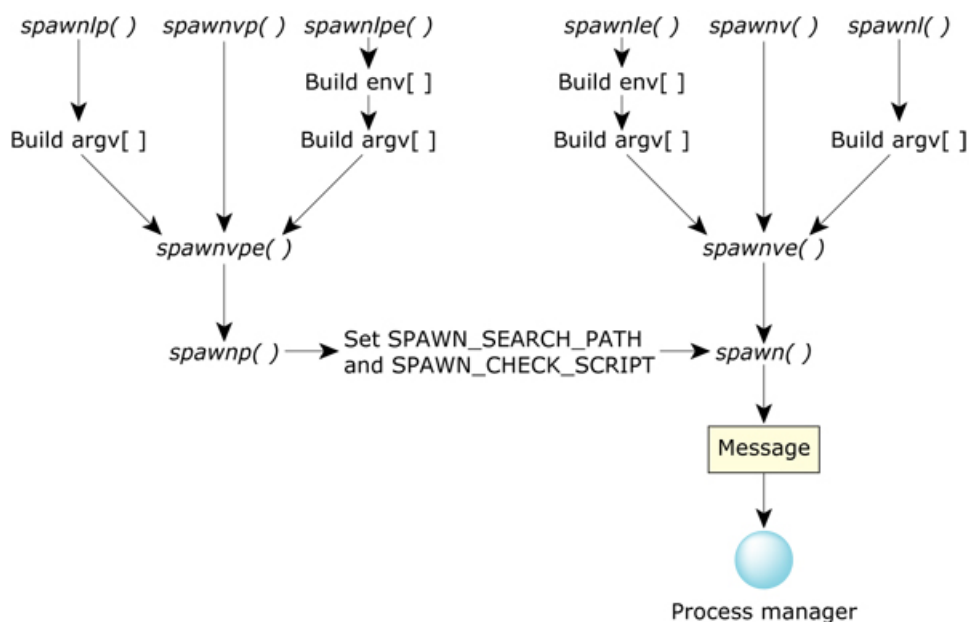
- In order to create a child process, your process must have the `PROCmgr_AID_SPAWN` ability enabled. For more information, see `procmgr_ability()` (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnvp()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It calls `spawnvpe()` before calling `spawnp()`. For greater portability, use `posix_spawn()` (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



The child process inherits the parent's environment. The environment is the collection of environment variables whose values that have been defined with the `export` shell command, the `env` utility, or by the successful execution of the `putenv()` (p. 2454) or

[setenv\(\)](#) (p. 2853) function. A program may read these values with the [getenv\(\)](#) (p. 838) function.



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The *spawnvp()* function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract information from this status, see “ Status macros (p. 3652)” in the documentation for <i>wait()</i> .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote node. You cannot get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned ([errno](#) (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process's file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is P_WAIT, and the spawned process terminated before the call to *waitpid()* was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is NULL, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *file* and its path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *file* argument is empty, or one or more components of the pathname of the new process don't exist.

ENOEXEC

The child process file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnvp()* function isn't implemented for the filesystem specified in *file*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set `PROT_EXEC` for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is `P_WAIT`, this function is a cancellation point.

spawnvpe()

Spawn a child process, given a vector of arguments, an environment, and a relative path

Synopsis:

```
#include <spawn.h>

int spawnvpe( int mode,
               const char * file,
               char * const argv[],
               char * const envp[] );
```

Arguments:

mode

How you want to load the child process, and how you want the parent program to behave after the child program is initiated:

- **P_WAIT** — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
- **P_NOWAIT** — execute the parent program concurrently with the new child process.
- **P_NOWAITO** — execute the parent program concurrently with the new child process. You can't use [wait\(\)](#) (p. 3652) to obtain the exit code.
- **P_OVERLAY** — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

file

The name of the executable file. If this argument contains a slash, it's used as the pathname of the executable; otherwise, the function searches for *file* in the directories listed in the **PATH** environment variable.

argv

A pointer to an argument vector; this argument can't be **NULL**. The value in `argv[0]` can't be **NULL**, and should represent the filename of the program being loaded. The last member of *argv* must be a **NULL** pointer.

envp

NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `spawnvpe()` function creates and executes a new child process, named in *file* with the NULL-terminated list of arguments in the *argv* vector.

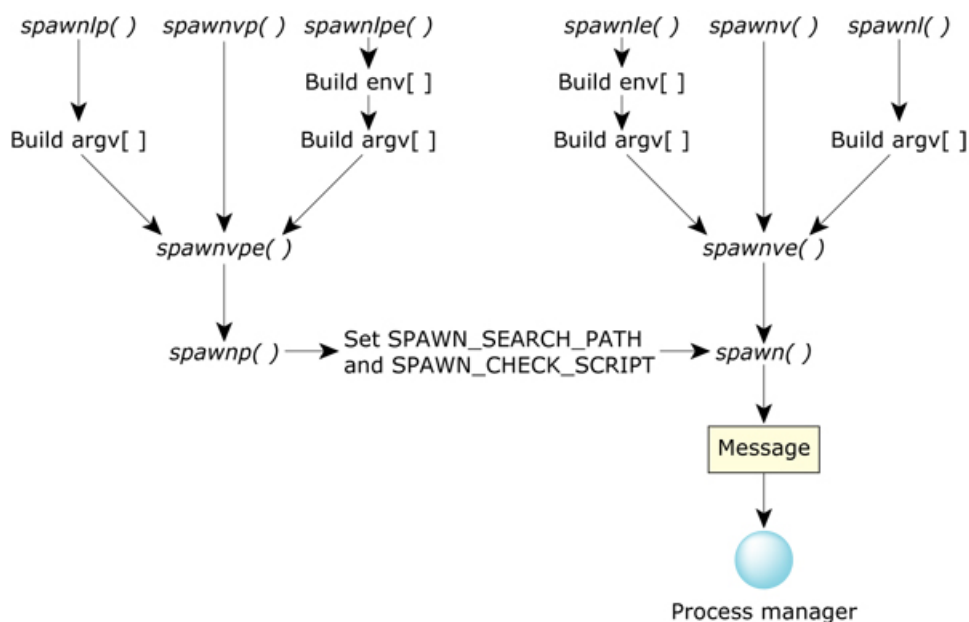
- In order to create a child process, your process must have the `PROC_MGR_AID_SPAWN` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).



- If the new child process is a shell script, the first line must start with `#!`, followed by the path of the program to run to interpret the script, optionally followed by one argument. The script must also be marked as executable. For more information, see “The first line” in the Writing Shell Scripts chapter of the QNX Neutrino *User's Guide*.

The `spawnvpe()` function *isn't* a POSIX 1003.1 function, and isn't guaranteed to behave the same on all operating systems. It calls `spawnp()`. For greater portability, use [posix_spawn\(\)](#) (p. 2037).

Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`:



If the value of *envp* is NULL, then the child process inherits the environment of the parent process. The new process can access its environment by using the [environ](#) (p. 528) global variable (found in <unistd.h>).

The following arguments are passed to the underlying call (*spawnp()*):

spawnvpe(mode, file, argv, envp)

- Map attribute flags.

Create temporary attribute structure and set the flags member based on the *mode*:

- 0 if *mode* = P_WAIT/P_NOWAIT
- SPAWN_EXEC if *mode* = P_OVERLAY
- SPAWN_NOZOMBIE if *mode* = P_NOWAITO
- Call *spawnp*(file, 0, 0, attr, argv, envp).



A parent/child relationship doesn't imply that the child process dies when the parent process dies.

Returns:

The *spawnvpe()* function's return value depends on the *mode* argument:

<i>mode</i>	Return value
P_WAIT	The exit status of the child process. For information about macros that extract

<i>mode</i>	Return value
	information from this status, see “ Status macros (p. 3652)” in the documentation for wait() .
P_NOWAIT	The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() (p. 3666) function, giving it this process ID.
P_NOWAITO	The process ID of the child process, or 0 if the process is being started on a remote node. You cannot get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned ([errno](#) (p. 539) is set).

Errors:

E2BIG

The number of bytes used by the argument list or environment list of the new child process is greater than ARG_MAX bytes.

EACCES

Search permission is denied for a directory listed in the path prefix of the new child process or the new child process file doesn't have the execute bit set.

EAGAIN

Insufficient resources available to create the child process.

EBADF

An error occurred duplicating open file descriptors to the new process.

ECHILD

The *mode* is P_WAIT, and the spawned process terminated before the call to [waitpid\(\)](#) was completed.

EFAULT

One of the buffers specified in the function call is invalid.

EINTR

The function was interrupted by a signal.

EINVAL

An argument is invalid (e.g. *arg0* is NULL, or the value of *mode* isn't valid).

ELOOP

Too many levels of symbolic links or prefixes.

EMFILE

Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of *file* plus its path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT

The file identified by the *file* argument is empty, or one or more components of the pathname of the child process don't exist.

ENOEXEC

The child process file has the correct permissions, but isn't in the correct format for an executable.

ENOMEM

Insufficient memory available to create the child process.

ENOSYS

The *spawnvpe()* function isn't implemented for the filesystem specified in *file*.

ENOTDIR

A component of the path prefix of the child process isn't a directory.

EPERM

The calling process doesn't have the required permission (see [procmgr_ability\(\)](#) (p. 2141)), or an underlying call to [mmap\(\)](#) (p. 1656) failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	Read the <i>Caveats</i>
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

If *mode* is P_WAIT, this function is a cancellation point.

sprintf()

Print formatted output into a string

Synopsis:

```
#include <stdio.h>

int sprintf( char* buf,
            const char* format,
            ... );
```

Arguments:

buf

A pointer to the buffer where you want the function to store the formatted string.

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `sprintf()` function is similar to [fprintf\(\)](#) (p. 702), except that `sprintf()` places the generated output into the character array pointed to by *buf*, instead of writing it to a file. A null character is placed at the end of the generated character string.

Returns:

The number of characters written into the array, not counting the terminating null character. An error can occur while converting a value for output. When an error occurs, [errno](#) (p. 539) indicates the type of error detected.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
```

```
/* Create temporary file names using a counter */
char namebuf[13];
int TempCount = 0;

char* make_temp_name()
{
    sprintf( namebuf, "ZZ%.6o.TMP", TempCount++ );
    return( namebuf );
}

int main( void )
{
    FILE* tf1,* tf2;

    tf1 = fopen( make_temp_name(), "w" );
    tf2 = fopen( make_temp_name(), "w" );
    fputs( "temp file 1", tf1 );
    fputs( "temp file 2", tf2 );
    fclose( tf1 );
    fclose( tf2 );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call this function in a signal handler if the data isn't floating point.

sqrt(), sqrtf(), sqrtl()

Calculate the nonnegative square root of a number

Synopsis:

```
#include <math.h>

double sqrt( double x );
float sqrtf( float x );
long double sqrtl( long double x );
```

Arguments:

x

The number that you want to calculate the square root of.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the nonnegative square root of *x*. A domain error occurs if the argument is negative.

Returns:

The nonnegative square root of the given number. For finite values of *x* < -0, these functions return NaN. The return value when *x* is -Inf is NaN.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set *errno* (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", sqrt(.5) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
0.707107
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

srand()

Start a new sequence of pseudo-random integers

Synopsis:

```
#include <stdlib.h>

void srand( unsigned int seed );
```

Arguments:

seed

The seed of the sequence of pseudo-random integers.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `srand()` function uses the argument *seed* to start a new sequence of pseudo-random integers to be returned by subsequent calls to [rand\(\)](#) (p. 2484). A particular sequence of pseudo-random integers can be repeated by calling `srand()` with the same *seed* value. The default sequence of pseudo-random integers is selected with a *seed* value of 1.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    int i;

    srand( 982 );
    for( i = 1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }

    /* Start the same sequence over again. */

    srand( 982 );
    for( i = 1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }

    /*
    Use the current time as a seed to
```

```
    get a different sequence.
*/

srand( (int) time( NULL ) );
for( i = 1; i < 10; ++i ) {
    printf( "%d\n", rand() );
}

return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

srand48()*Initialize a sequence of pseudo-random numbers***Synopsis:**

```
#include <stdlib.h>

void srand48( long seed );
```

Arguments:***seed***

The seed of the sequence of pseudo-random integers.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `srand48()` is used to initialize the internal buffer $r(n)$ of `drand48()`, `lrand48()`, and `mrnd48()` such that the 32 bits of the seed value are copied into the upper 32 bits of $r(n)$, with the lower 16 bits of $r(n)$ arbitrarily being set to `0x330E`. Additionally, the constant multiplicand and addend of the algorithm are reset to the default values: the multiplicand $a = 0xFDEECE66D = 25214903917$ and the addend $c = 0xB = 11$.

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

srandom()

Set the seed for a pseudo-random number generator

Synopsis:

```
#include <stdlib.h>

void srandom( unsigned int seed );
```

Arguments:

seed

The seed of the sequence of pseudo-random integers.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *srandom()* function initializes the current state array using the value of *seed*.

Use this function in conjunction with the following:

[*initstate\(\)*](#) (p. 1133)

Initialize the state of the pseudo-random number generator.

[*random\(\)*](#) (p. 2487)

Generate a pseudo-random number using a default state.

[*setstate\(\)*](#) (p. 2913)

Specify the state of the pseudo-random number generator.

The *random()* and *srandom()* functions have (almost) the same calling sequence and initialization properties as [*rand\(\)*](#) (p. 2484) and [*srand\(\)*](#) (p. 3155) Unlike *srand()*, *srandom()* doesn't return the old seed because the amount of state information used is much more than a single word. The *initstate()* and *setstate()* routines are provided to deal

with restarting/changing random number generators. With 256 bytes of state information, the period of the random-number generator is greater than 2^{69} .

Like *rand()*, *random()* produces by default a sequence of numbers that can be duplicated by calling *srandom()* with 1 as the seed.

After initialization, a state array can be restarted at a different point in one of two ways:

- The *initstate()* function can be used, with the desired seed, state array, and size of the array.
- The *setstate()* function, with the desired state, can be used, followed by *srandom()* with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

_srealloc()

Allocate, reallocate or free a block of memory

Synopsis:

```
#include <malloc.h>

void *_srealloc( void* ptr,
                 size_t old_size,
                 size_t new_size );
```

Arguments:

ptr

NULL, or a pointer to the block of memory that you want to reallocate.

old_size

The current size of the block, in bytes.

new_size

The size of the block to allocate, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

When the value of the *ptr* argument is NULL, a new block of memory of *new_size* bytes is allocated.

If the value of *new_size* is zero, the corresponding `_sfree()` function is called to release *old_size* bytes of memory memory pointed to by *ptr*.

Otherwise, the `_srealloc()` function reallocates space for an object of *new_size* bytes by doing one of the following:

- Shrinking the allocated size of the allocated memory block *ptr* when *new_size* is sufficiently smaller than *old_size*.

Or:

- Extending the allocated size of the allocated memory block *ptr* if there is a large enough block of unallocated memory immediately following *ptr*.

Or:

- Allocating a new block, and copying the contents of *ptr* to the new block.



Because it's possible that a new block will be allocated, don't maintain any pointers into the old memory after a successful call to this function. These pointers will point to freed memory, with possible disastrous results when a new block is allocated.

The function returns `NULL` when the memory pointed to by *ptr* can't be reallocated. In this case, the memory pointed to by *ptr* isn't freed, so be sure to keep a pointer to the old memory block.

```
buffer = (char *) _srealloc( buffer, 100, 200 );
```

In the above example, *buffer* is set to `NULL` if the function fails, and no longer points to the old memory block. If *buffer* is your only pointer to the memory block, then you've lost access to this memory.

The `_srealloc()` function reallocates memory from the heap.

You must use `_sfree()` (p. 2923) to deallocate the memory allocated by `_srealloc()`.

Returns:

A pointer to the start of the reallocated memory, or `NULL` if there's insufficient memory available, or if the value of the *new_size* argument is zero.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

sscanf()

Scan input from a character string

Synopsis:

```
#include <stdio.h>

int sscanf( const char* in_string,
            const char* format,
            ... );
```

Arguments:

in_string

The string that you want to read from.

format

A string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sscanf()` function scans input from the character string *in_string*, under control of the argument *format*. Following the format string is the list of addresses of items to receive values.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is terminated by reaching the end of the input string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int day, year;
```

```

char weekday[20], month[20];

sscanf( "Thursday February 0025 1999",
        "%s %s %d %d",
        weekday, month, &day, &year );
printf( "%s %d, %d is a %s\n",
        month, day, year, weekday );
return EXIT_SUCCESS;
}

```

produces the following:

```
February 25, 1999 is a Thursday
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call this function in a signal handler if the data isn't floating point.

stat()*, *stat64()

Get information about a file or directory, given a path

Synopsis:

```
#include <sys/stat.h>

int stat( const char * path,
          struct stat * buf );

int stat64( const char * path,
            struct stat64 * buf );
```

Arguments:

path

The path of the file or directory that you want information about.

buf

A pointer to a buffer where the function can store the information; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *stat()* and *stat64()* functions obtain information about the file or directory referenced in *path*. This information is placed in the structure located at the address indicated by *buf*. The *stat64()* function is a 64-bit version of *stat()*.

stat structure

Here's the *stat* structure that's defined in `<sys/stat.h>`:

```
struct stat {
#ifdef _FILE_OFFSET_BITS - 0 == 64
    ino_t      st_ino;           /* File serial number. */
    off_t      st_size;         /* File size in bytes. */
#elif defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
#ifdef __LITTLEENDIAN__
    ino_t      st_ino;           /* File serial number. */
    ino_t      st_ino_hi;
    off_t      st_size;
    off_t      st_size_hi;
#elif defined(__BIGENDIAN__)
    ino_t      st_ino_hi;
    ino_t      st_ino;           /* File serial number. */
    off_t      st_size_hi;
    off_t      st_size;
#else
    #error endian not configured for system
#endif
#endif
};
```



```

#error _FILE_OFFSET_BITS value is unsupported
#endif
dev_t      st_dev;      /* ID of the device containing the file. */
dev_t      st_rdev;     /* Device ID. */
uid_t      st_uid;      /* User ID of file. */
gid_t      st_gid;      /* Group ID of file. */
time_t     st_mtime;    /* Time of last data modification. */
time_t     st_atime;    /* Time when file data was last accessed. */
time_t     st_ctime;    /* Time of last file status change. */
mode_t     st_mode;     /* File types and permissions. */
nlink_t    st_nlink;    /* Number of hard links to the file. */
blksize_t  st_blocks;   /* Size of a block used by st_nblocks. */
int32_t    st_nblocks;  /* Number of blocks st_blocks blocks. */
blksize_t  st_blksize;  /* Preferred I/O block size for object. */
#if _FILE_OFFSET_BITS - 0 == 64
    blkcnt_t st_blocks; /* No. of 512-byte blocks allocated for a file. */
#elif defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
    blkcnt_t st_blocks; /* No. of 512-byte blocks allocated for a file. */
    blkcnt_t st_blocks_hi;
#elif defined(__BIGENDIAN__)
    blkcnt_t st_blocks_hi;
    blkcnt_t st_blocks;
#else
#error endian not configured for system
#endif
#endif
#error _FILE_OFFSET_BITS value is unsupported
#endif
};

```

Access permissions

The access permissions for the file or directory are specified as a combination of bits in the *st_mode* field of a *stat* structure. These bits are defined in `<sys/stat.h>`, and are described below:

Owner	Group	Others	Permission
S_IRUSR	S_IRGRP	S_IROTH	Read
S_IRWXU	S_IRWXG	S_IRWXO	Read, write, execute/search; a bitwise inclusive OR of the other three constants. (S_IRWXU is an OR of IRUSR, S_IWSUR and S_IXUSR.)
S_IWUSR	S_IWGRP	S_IWOTH	Write
S_IXUSR	S_IXGRP	S_IXOTH	Execute/search

The following bits define miscellaneous permissions used by other implementations:

Bit	Equivalent
S_IEXEC	S_IXUSR
S_IREAD	S_IRUSR
S_IWRITE	S_IWUSR

st_mode bits

The following bits are also encoded in the *st_mode* field:

S_ISUID

Set user ID on execution. The process's effective user ID is set to that of the owner of the file when the file is run as a program. On a regular file, this bit should be cleared on any write.

S_ISGID

Set group ID on execution. Set effective group ID on the process to the file's group when the file is run as a program. On a regular file, this bit should be cleared on any write.

File types

The following bits in *st_mode* identify the file type:

S_IFBLK

Block special.

S_IFCHR

Character special.

S_IFDIR

Directory.

S_IFIFO

FIFO special.

S_IFLNK

Symbolic link.

S_IFNAM

Special named file.

S_IFREG

Regular.

S_IFSOCK

Socket.

S_IFMT is a mask that isolates the above bits.

Macros

The following macros test whether a file is of a specified type. The value *m* supplied to the macros is the value of the *st_mode* field of a *stat* structure. The macros evaluate to a nonzero value if the test is true, and zero if the test is false.

S_ISBLK(m)

Test for block special file.

S_ISCHR(m)

Test for character special file.

S_ISDIR(m)

Test for directory file.

S_ISFIFO(m)

Test for FIFO.

S_ISLNK(m)

Test for symbolic link.

S_ISNAM(m)

Test for special named file.

S_ISREG(m)

Test for regular file.

S_ISSOCK(m)

Test for socket.

These macros test whether a file is of the specified type. The value of the *buf* argument supplied to the macros is a pointer to a `stat` structure. The macro evaluates to a nonzero value if the specified object is implemented as a distinct file type and the specified file type is contained in the `stat` structure referenced by the pointer *buf*. Otherwise, the macro evaluates to zero.

S_TYPEISMQ(buf)

Test for message queue.

S_TYPEISSEM(buf)

Test for semaphore.

S_TYPEISSHM(buf)

Test for shared memory object.

These macros manipulate device IDs:

major(device)

Extract the major number from a device ID.

minor(device)

Extract the minor number from a device ID.

makedev(node, major, minor)

Build a device ID from the given numbers. Currently, the *node* argument isn't used and must be zero.

The *st_rdev* member of the *stat* structure is a device ID that consists of:

- a major number in the range 0 through 63
- a minor number in the range 0 through 1023.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

Search permission is denied for a component of *path*.

EIO

A physical error occurred on the block device.

ELOOP

Too many levels of symbolic links or prefixes.

ENAMETOOLONG

The argument *path* exceeds `PATH_MAX` in length, or a pathname component is longer than `NAME_MAX`. These manifests are defined in the `<limits.h>` header file.

ENOENT

The named file doesn't exist, or *path* is an empty string.

ENOSYS

The *stat()* function isn't implemented for the filesystem specified in *path*.

ENOTDIR

A component of *path* isn't a directory.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number can't be represented correctly in the structure pointed to by *buf*.

Examples:

Determine the size of a file:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main( void )
{
    struct stat buf;

    if( stat( "file", &buf ) != -1 ) {
        printf( "File size = %d\n", buf.st_size );
    }
    return EXIT_SUCCESS;
}
```

Determine the amount of free memory:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main () {
    struct stat buf;

    if ( stat( "/proc", &buf ) == -1) {
        perror ( "stat" );
        return EXIT_FAILURE;
    } else {
        printf ( "Free memory: %d bytes\n", buf.st_size);
        return EXIT_SUCCESS;
    }
}
```

Classification:

stat() is [POSIX 1003.1](#) (p. 46); *stat64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

statvfs(), statvfs64()

Get filesystem information, given a path

Synopsis:

```
#include <sys/statvfs.h>

int statvfs( const char *path,
             struct statvfs *buf );

int statvfs64( const char *path,
               struct statvfs64 *buf );
```

Arguments:

path

The name of a file that resides on the filesystem.

buf

A pointer to a buffer where the function can store the information.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *statvfs()* and *statvfs64()* functions return a “generic superblock” describing a filesystem that you can use to acquire information about mounted filesystems. The *statvfs64()* function is a 64-bit version of *statvfs()*. The [fstatvfs\(\)](#) and [fstatvfs64\(\)](#) (p. 783) functions are similar, but they take a file descriptor instead of a path.

The filesystem type is known to the operating system. You don't need to have read, write, or execute permission for the named file, but all directories listed in the path name leading to the file must be searchable.

The *buf* argument is a pointer to a *statvfs* or *statvfs64* structure that's filled by the function. It contains at least:

unsigned long *f_bsize*

The preferred filesystem blocksize.

unsigned long *f_fsize*

The fundamental filesystem blocksize (if supported)

fsblkcnt_t *f_blocks*

The total number of blocks on the filesystem, in units of *f_fsize*.

fsblkcnt_t *f_bfree*

The total number of free blocks.

fsblkcnt_t *f_bavail*

The number of free blocks available to a nonsuperuser.

fsfilcnt_t *f_files*

The total number of file nodes (inodes).

fsfilcnt_t *f_ffree*

The total number of free file nodes.

fsfilcnt_t *f_favail*

The number of inodes available to a nonsuperuser.

unsigned long *f_fsid*

The filesystem ID (currently the device ID).

char *f_basetype*[16]

The type of the target filesystem, as a null-terminated string.

unsigned long *f_flag*

A bitmask of flags; the function can set these flags (the `_MOUNT_*` and `ST_*` bits are equivalent):

<code>_MOUNT_*</code>	<code>ST_*</code>	Meaning
<code>_MOUNT_READONLY</code>	<code>ST_RDONLY</code>	The filesystem is read-only.
<code>_MOUNT_NOEXEC</code>	<code>ST_NOEXEC</code>	The filesystem doesn't permit the loading of executables.
<code>_MOUNT_NOSUID</code>	<code>ST_NOSUID</code>	The filesystem doesn't support setuid() (p. 2917)

<code>_MOUNT_*</code>	<code>ST_*</code>	Meaning
		and setgid() (p. 2858) semantics.
<code>_MOUNT_NOCREAT</code>	<code>ST_NOCREAT</code>	You can't create files on the filesystem.
<code>_MOUNT_OFF32</code>	<code>ST_OFF32</code>	The <code>off_t</code> type is limited to 32 bits.
<code>_MOUNT_NOATIME</code>	<code>ST_NOATIME</code>	The filesystem doesn't support the logging of file access times.

unsigned long *f_namemax*

The maximum filename length.

The values returned for *f_files*, *f_ffree*, and *f_favail* depend on the filesystem:

fs-dos.so, fs-qnx4.so

These filesystems embed inodes in directories, so the number of files is limited by the amount of disk space, so basically there's no limit. They don't count the number of files created either, so they set all these fields to 0.

fs-qnx6.so, fs-etfs-ram, fs-ext2.so

These filesystems have a fixed inodes table, so they fill in these fields.

fs-cd.so, fs-mac.so, fs-nt.so, fs-udf.so

Read-only fsystems always set *f_ffree* and *f_favail* to 0. Where possible or known, they try to set *f_files*:

- *fs-mac.so* knows exactly
- *fs-nt.so* provides an upper bound based on the current size of the Master File Table (MFT)
- *fs-udf.so* knows when it's a UDF that's mounted; ISO formats don't indicate that information, so *f_files* is set to 0
- *fs-cd.so* sets it to 0

fs-cifs

The CIFS filesystem sets these fields to 0.

devf-*

Flash filesystem drivers estimate the values of these fields, based on the amount of free space.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied on a component of the path prefix.

EFAULT

The *path* or *buf* argument points to an illegal address.

EINTR

A signal was caught during execution.

EIO

An I/O error occurred while reading the filesystem.

ELOOP

Too many symbolic links were encountered in translating *path*.

EMULTIHOP

Components of path require hopping to multiple remote machines and the filesystem type doesn't allow it.

ENAMETOOLONG

The length of a path component exceeds **{NAME_MAX}** characters, or the length of *path* exceeds **{PATH_MAX}** characters.

ENOENT

Either a component of the path prefix or the file referred to by *path* doesn't exist.

ENOLINK

The *path* argument points to a remote machine and the link to that machine is no longer active.

ENOTDIR

A component of the path prefix of *path* isn't a directory.

EOverflow

One of the values to be returned can't be represented correctly in the structure pointed to by *buf*.

Classification:

statvfs() is [POSIX 1003.1 XSI](#) (p. 46); *statvfs64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

stderr

The standard error stream

Synopsis:

```
#include <stdio.h>

FILE * stderr;
```

Description:

This global variable defines the standard error stream. It's set to the console by default, but you can redirect it by calling [freopen\(\)](#) (p. 721).

STDERR_FILENO, which is defined in `<unistd.h>`, defines the file descriptor that corresponds to *stderr*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

stdin

The standard input stream

Synopsis:

```
#include <stdio.h>

FILE * stdin;
```

Description:

This global variable defines the standard input stream. It's set to the console by default, but you can redirect it by calling [freopen\(\)](#) (p. 721).

STDIN_FILENO, which is defined in `<unistd.h>`, defines the file descriptor that corresponds to *stdin*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

stdout

The standard output stream

Synopsis:

```
#include <stdio.h>

FILE * stdout;
```

Description:

This global variable defines the standard output stream. It's set to the console by default, but you can redirect it by calling [freopen\(\)](#) (p. 721).

STDOUT_FILENO, which is defined in <unistd.h>, defines the file descriptor that corresponds to *stdout*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

straddstr()

Concatenate one string on to the end of another

Synopsis:

```
#include <string.h>

int straddstr( const char * str,
               int len,
               char ** pbuf,
               size_t * pmaxbuf );
```

Arguments:

str

The string that you want to add to the end of another.

len

The number of characters from *str* that you want to add. If zero, the function adds all of *str*.

pbuf

The address of a pointer to the destination buffer.

pmaxbuf

A pointer to the size of the destination buffer.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *straddstr()* function adds *str* to the buffer pointed to by *pbuf*, respecting the maximum length indicated by *pmaxbuf*, and updates the values of *pbuf* and *pmaxbuf*:

- If *len* is zero, *straddstr()* calls [strlen\(\)](#) (p. 3213) to determine the length of *str*. In this case, *str* must be null-terminated.
- The *straddstr()* function null-terminates *pbuf* whenever *pmaxbuf* is nonzero, even if the source string isn't null-terminated (but *len* must be nonzero in this case).

- If *len* is nonzero, null bytes in the input have no special meaning (i.e., *straddstr()* doesn't stop copying when it reads a null byte, but it still null-terminates *pbuf*).
- If the full string can't be copied, it's truncated, but *pbuf* is still null-terminated.

Returns:

The value of *len* if it's nonzero; otherwise, the length of *str* (i.e. `strlen(str)`).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcasecmp()

Compare two strings, ignoring case

Synopsis:

```
#include <strings.h>

int strcasecmp( const char* str1,
                const char* str2 );
```

Arguments:

str1, str2

The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcasecmp()* function compares two strings, specified by *str1* and *str2*, ignoring the case of the characters.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>

void compare( const char* s1, const char* s2 )
{
```



```

    int retval;

    retval = strcasecmp( s1, s2 );
    if( retval > 0 ) {
        printf( "%s > %s\n", s1, s2 );
    } else if( retval < 0 ) {
        printf( "%s < %s\n", s1, s2 );
    } else {
        printf( "%s == %s\n", s1, s2 );
    }
}

int main( void )
{
    char* str1 = "abcdefg";
    char* str2 = "HIJ";
    char* str3 = "Abc";
    char* str4 = "aBCDEfg";

    compare( str1, str2 );
    compare( str1, str3 );
    compare( str1, str4 );
    compare( str1, str1 );

    compare( str2, str2 );
    compare( str2, str3 );
    compare( str2, str4 );
    compare( str2, str1 );

    return EXIT_SUCCESS;
}

```

This code produces output that looks like:

```

abcdefg < HIJ
abcdefg > Abc
abcdefg == aBCDEfg
abcdefg == abcdefg
HIJ == HIJ
HIJ > Abc
HIJ > aBCDEfg
HIJ > abcdefg

```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcat()

Concatenate two strings

Synopsis:

```
#include <string.h>

char* strcat( char* dst,
              const char* src );
```

Arguments:

dst, src

The strings that you want to concatenate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcat()* function appends a copy of the string pointed to by *src* (including the terminating NUL character) to the end of the string pointed to by *dst*. The first character of *src* overwrites the NUL character at the end of *dst*.

Returns:

The same pointer as *dst*.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];

    strcpy( buffer, "Hello " );
    strcat( buffer, "world" );

    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}
```

produces the output:

```
Hello world
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strchr()

Find the first occurrence of a character in a string

Synopsis:

```
#include <string.h>

char* strchr(char* s,
             int c );
```

Arguments:

s

The string that you want to search.

c

The character that you're looking for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strchr()* function finds the first occurrence of *c* (converted to a `char`) in the string pointed to by *s*. The terminating NUL character is considered to be part of the string.

Returns:

A pointer to the located character, or `NULL` if *c* doesn't occur in the string.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
    char* where;

    strcpy( buffer, "video x-rays" );
    where = strchr( buffer, 'x' );
    if( where == NULL ) {
```

```
        printf( "'x' not found\n" );
    } else {
        printf( "'x' found: %s\n", where );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcmp()

Compare two strings

Synopsis:

```
#include <string.h>

int strcmp( const char* s1,
            const char* s2 );
```

Arguments:

s1, s2

The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcmp()* function compares the string pointed to by *s1* to the string pointed to by *s2*.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
```

```

    printf( "%d\n", strcmp( "abcdef", "abcdef" ) );
    printf( "%d\n", strcmp( "abcdef", "abc" ) );
    printf( "%d\n", strcmp( "abc", "abcdef" ) );
    printf( "%d\n", strcmp( "abcdef", "mnopqr" ) );
    printf( "%d\n", strcmp( "mnopqr", "abcdef" ) );

    return EXIT_SUCCESS;
}

```

produces the output:

```

0
1
-1
-1
1

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcmpi()

Compare two strings, ignoring case

Synopsis:

```
#include <string.h>

int strcmpi( const char* s1,
             const char* s2 );
```

Arguments:

s1, s2

The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcmpi()* function compares the string pointed to by *s1* to the string pointed to by *s2*, ignoring case.

All uppercase characters from *s1* and *s2* are mapped to lowercase for the purposes of doing the comparison. The *strcmpi()* function is identical to the [stricmp\(\)](#) (p. 3208) function.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", strcmpi( "AbCDEF", "abcdef" ) );
    printf( "%d\n", strcmpi( "abcdef", "ABC" ) );
    printf( "%d\n", strcmpi( "abc", "ABCdef" ) );
    printf( "%d\n", strcmpi( "Abcdef", "mnopqr" ) );
    printf( "%d\n", strcmpi( "Mnopqr", "abcdef" ) );

    return EXIT_SUCCESS;
}

```

produces the output:

```

0
100
-100
-12
12

```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcoll()

Compare two strings, using the locale's collating sequence

Synopsis:

```
#include <string.h>

int strcoll( const char* s1,
             const char* s2 );
```

Arguments:

s1, s2

The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcoll()* function compares the strings pointed to by *s1* and *s2*, using the collating sequence selected by the [setlocale\(\)](#) (p. 2880) function.

The *strcoll()* function is equivalent to [strcmp\(\)](#) (p. 3186) when the collating sequence is selected from the "C" locale.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

char buffer[80] = "world";

int main( void )
{
    if( strcoll( buffer, "Hello" ) < 0 ) {
        printf( "Less than\n" );
    }

    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcpy()

Copy a string

Synopsis:

```
#include <string.h>

char* strcpy( char* dst,
              const char* src );
```

Arguments:

dst

A pointer to where you want to copy the string.

src

The string that you want to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcpy()* function copies the string pointed to by *src* (including the terminating NUL character) into the array pointed to by *dst*.



Copying of overlapping objects isn't guaranteed to work properly. See the [memmove\(\)](#) (p. 1618) function for information on copying objects that overlap.

Returns:

The same pointer as *dst*.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
```

```
    strcpy( buffer, "Hello " );  
    strcat( buffer, "world" );  
  
    printf( "%s\n", buffer );  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
Hello world
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strcspn()

Count the characters at the beginning of a string that aren't in a given character set

Synopsis:

```
#include <string.h>

size_t strcspn( const char* str,
                const char* charset );
```

Arguments:

str

The string that you want to search.

charset

The set of characters you want to look for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strcspn()* function finds the length of the initial segment of the string pointed to by *str* that consists entirely of characters *not* from the string pointed to by *charset*. The terminating NUL character isn't considered part of *str*.

Returns:

The length of the initial segment.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", strcspn( "abcbcadef", "cba" ) );
    printf( "%d\n", strcspn( "xxxbcadef", "cba" ) );
    printf( "%d\n", strcspn( "123456789", "cba" ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0  
3  
9
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strdup()

Create a duplicate of a string

Synopsis:

```
#include <string.h>

char* strdup( const char* src );
```

Arguments:

src

The string that you want to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strdup()* function creates a duplicate of the string pointed to by *src*, and returns a pointer to the new copy.



The *strdup()* function allocates the memory for the new string by calling [malloc\(\)](#) (p. 1569); it's up to you to release the memory by calling [free\(\)](#) (p. 717).

Returns:

A pointer to a copy of the string, or `NULL` if an error occurred.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char *dup;

    dup = strdup( "Make a copy" );
    printf( "%s\n", dup );
    free (dup);

    return EXIT_SUCCESS;
}
```


Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

strerror()

Convert an error number into an error message

Synopsis:

```
#include <string.h>

char* strerror( int errnum );
```

Arguments:

errnum

The error number that you want the message for. This function works for any valid [errno](#) (p. 539) value.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *strerror()* function maps the error number contained in *errnum* to an error message.

Returns:

A pointer to the error message.



Don't modify the string that this function returns.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;

    fp = fopen( "file.name", "r" );
    if( fp == NULL ) {
        printf( "Unable to open file: %s\n",
               strerror( errno ) );
    }
}
```

```
    return EXIT_SUCCESS;  
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

strerror_r()

Convert an error number into an error message (reentrant)

Synopsis:

```
#include <string.h>

int strerror_r( int errnum,
               char *strerrbuf,
               size_t buflen );
```

Arguments:

errnum

The error number that you want the message for. This function works for any valid [errno](#) (p. 539) value.

strerrbuf

A pointer to a buffer where the function can store the error message.

buflen

The length of the buffer, in bytes.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *strerror_r()* function maps the error number contained in *errnum* to an error message, which it stores in the buffer that *strerrbuf* points to.

Returns:

0

Success.

EINVAL

The value of *errnum* isn't a valid error number.

ERANGE

The buffer pointed to by *strerrbuf* isn't big enough to hold the generated message string.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

#define MSG_LEN 100

int main( void )
{
    FILE *fp;
    char msg_buff[MSG_LEN];
    int error_num;

    fp = fopen( "file.name", "r" );
    if( fp == NULL ) {
        error_num = strerror_r ( errno, msg_buff, MSG_LEN);

        switch (error_num) {
            case 0:
                printf( "Unable to open file: %s\n", msg_buff);
                break;

            case EINVAL:
                printf ( "strerror_r() failed: invalid error code, %d\n",
                        error_num);
                break;

            case ERANGE:
                printf ( "strerror_r() failed: buffer too small: %d\n",
                        MSG_LEN);
                break;

        }
    }
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

strftime()

Format a time into a string

Synopsis:

```
#include <time.h>

size_t strftime( char * s,
                 size_t maxsize,
                 const char * format,
                 const struct tm * timeptr );
```

Arguments:

s

A pointer to a buffer where the function can store the formatted time.

maxsize

The maximum size of the buffer.

format

The format that you want to use for the time; see “[Formats](#) (p. 3202),” below.

timeptr

A pointer to a [tm](#) (p. 3476) structure that contains the time that you want to format.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *strftime()* function formats the time in the argument *timeptr* into the array pointed to by the argument *s*, according to the *format* argument.

Formats

The *format* string consists of zero or more directives and ordinary characters. A directive consists of a % character followed by a character that determines the substitution

that's to take place. All ordinary characters are copied unchanged into the array. No more than *maxsize* characters are placed in the array.

Local timezone information is used as if from a call to [tzset\(\)](#) (p. 3547).

%a

Locale's abbreviated weekday name.

%A

Locale's full weekday name.

%b

Locale's abbreviated month name.

%B

Locale's full month name.

%c

Locale's appropriate date and time representation.

%d

Day of the month as a decimal number (01-31).

%D

Date in the format *mm/dd/yy*.

%e

Day of the month as a decimal number (1-31); single digits are preceded by a space.

%F

The ISO standard date format; equivalent to %Y-%m-%d.

%g

The last two digits of the week-based year as a decimal number (00-99).

%G

The week-based year as a decimal number (e.g. 1998).

%h

Locale's abbreviated month name.

%H

Hour (24-hour clock) as a decimal number (00-23).

%I

Hour (12-hour clock) as a decimal number (01-12).

%j

Day of the year as a decimal number (001-366).

%m

Month as a decimal number (01-12).

%M

Minute as a decimal number (00-59).

%n

Newline character.

%p

Locale's equivalent of either AM or PM.

%r

12-hour clock time (01-12) using the AM/PM notation in the format *HH:MM:SS* (AMIPM).

%R

24-hour notation; %H:%M.

%S

Second as a decimal number (00-59).

%t

Tab character.

%T

24-hour clock time in the format *HH:MM:SS*.

%u

Weekday as a decimal number (1-7), where Monday is 1.

%U

Week number of the year as a decimal number (00-52), where Sunday is the first day of the week.

%V

Week number of the year as a decimal number (01-53), where Monday is the first day of the week. The week containing January 1 is in the new year if four or more days are in the new year, otherwise it is the last week of the previous year.

%w

Weekday as a decimal number (0-6), where 0 is Sunday.

%W

Week number of the year as a decimal number (00-52), where Monday is the first day of the week.

%x

Locale's appropriate date representation.

%X

Locale's appropriate time representation.

%y

Year without century, as a decimal number (00-99).

%Y

Year with century, as a decimal number.

%z

Offset from UTC -0430 (4 hrs, 30 minutes behind UTC, west of Greenwich), or no characters if time zone isn't specified.

%Z

Time zone name, or no characters if time zone isn't specified.

%%

Character %.

Some of the above conversion specifiers can be modified with the prefix E or O. If alternative formats don't exist for the locale, they behave as if the unmodified conversion specifiers were called:

%Ec

Alternative date and time representation.

%EC

Alternative name of the the base year (period).

%Ex

Alternative date representation.

%EX

Alternative time representation.

%Ey

Offset from %EC of the alternative year (only) representation.

%EY

Alternative year representation.

%Od

Day of the month using alternative numeric symbols. Leading zeros are added if an alternative symbol for zero exists, otherwise leading spaces are used.

%Oe

Day of the month using alternative numeric symbols. Leading spaces are used.

%OH

24-hour clock using alternative numeric symbols.

%OI

12-hour clock using alternative numeric symbols.

%Om

Month using alternative numeric symbols.

%OM

Minutes using alternative numeric symbols.

%OS

Seconds using alternative numeric symbols.

%Ou

Alternative week day number representation (Monday=1).

%OU

Alternative week day number representation (Rules correspond with %U).

%OV

Alternative week number representation. (Rules correspond with %V).

%Ow

Weekday as a number using alternative numeric symbols (Sunday=0).

%OW

Week number of the year using alternative numeric symbols (Monday is the first day of the week).

%Oy

Year offset from %C using alternative numeric symbols.

Returns:

The number of characters placed into the array, not including the terminating null character, or 0 if the number of characters exceeds *maxsize*; the string contents are indeterminate.

If an error occurs, [errno](#) (p. 539) is set.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    time_t time_of_day;
    char buffer[ 80 ];

    time_of_day = time( NULL );
    strftime( buffer, 80, "Today is %A %B %d, %Y",
              localtime( &time_of_day ) );
    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}
```

This produces the output:

Today is Thursday February 25, 1999

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

stricmp()

Compare two strings, ignoring case

Synopsis:

```
#include <string.h>

int stricmp( const char* s1,
             const char* s2 );
```

Arguments:

s1, s2

The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *stricmp()* function compares, with case insensitivity, the string pointed to by *s1* to the string pointed to by *s2*. All uppercase characters from *s1* and *s2* are mapped to lowercase for the purposes of doing the comparison.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
int main( void )
{
    printf( "%d\n", stricmp( "AbCDEF", "abcdef" ) );
    printf( "%d\n", stricmp( "abcdef", "ABC" ) );
    printf( "%d\n", stricmp( "abc", "ABCdef" ) );
    printf( "%d\n", stricmp( "Abcdef", "mnopqr" ) );
    printf( "%d\n", stricmp( "Mnopqr", "abcdef" ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0
100
-100
-12
12
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strlcat()*, *strlcpy()

Size-bounded string copying and concatenation

Synopsis:

```
#include <string.h>

size_t strlcpy( char *dst,
               const char *src,
               size_t size );

size_t strlcat( char *dst,
               const char *src,
               size_t size );
```

Arguments:

dst

A pointer to the destination string.

src

A pointer to the source string.

size

The size of the destination buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strlcpy()* and *strlcat()* functions copy and concatenate strings respectively. They're designed to be safer, more consistent, and less error-prone replacements for [strncpy\(\)](#) (p. 3223) and [strncat\(\)](#) (p. 3219).

Unlike those functions, *strlcpy()* and *strlcat()* take the full size of the buffer (not just the length) and guarantee to NUL-terminate the result (as long as *size* is larger than 0 or, in the case of *strlcat()*, as long as there's at least one byte free in *dst*).



You should include a byte for the NUL in *size*. Also note that *strlcpy()* and *strlcat()* operate only on true “C” strings. This means that for *strlcpy()*, *src*

must be NUL-terminated, and for *strlcat()*, both *src* and *dst* must be NUL-terminated.

The *strlcpy()* function copies up to *size* – 1 characters from the NUL-terminated string *src* to *dst*, NUL-terminating the result.

The *strlcat()* function appends the NUL-terminated string *src* to the end of *dst*. It will append at most *size* – *strlen(dst)* – 1 bytes, NUL-terminating the result.

Returns:

The total length of the string:

- For *strlcpy()* that means the length of *src*.
- For *strlcat()* that means the initial length of *dst* plus the length of *src*. While this may seem somewhat confusing, it was done to make truncation detection simple.



If *strlcat()* traverses *size* characters without finding a NUL, the length of the string is considered to be *size*, and the destination string isn't NUL-terminated (since there was no space for the NUL). This keeps *strlcat()* from running off the end of a string. In practice this shouldn't happen (as it means that either *size* is incorrect or that *dst* isn't a proper "C" string). The check exists to prevent potential security problems in incorrect code.

Examples:

The following code fragment illustrates the simple case:

```
char *s, *p, buf[BUFSIZ];
...
(void)strlcpy(buf, s, sizeof(buf));
(void)strlcat(buf, p, sizeof(buf));
```

To detect truncation, perhaps while building a pathname, you could use something like this:

```
char *dir, *file, pname[MAXPATHLEN];
...
if (strlcpy(pname, dir, sizeof(pname)) >= sizeof(pname))
    goto toolong;
if (strlcat(pname, file, sizeof(pname)) >= sizeof(pname))
    goto toolong;
```

Since we know how many characters we copied the first time, we can speed things up a bit by using a copy instead of an append:

```
char *dir, *file, pname[MAXPATHLEN];
size_t n;
...
n = strlcpy(pname, dir, sizeof(pname));
if (n >= sizeof(pname))
```

```
        goto toolong;
if (strncpy(pname + n, file, sizeof(pname) - n) >=
    sizeof(pname) - n)
    goto toolong;
```

However, one may question the validity of such optimizations, as they defeat the whole purpose of *strncpy()* and *strncat()*.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Contributing author:

OpenBSD

strlen()

Compute the length of a string

Synopsis:

```
#include <string.h>

size_t strlen( const char * s );
```

Arguments:

s

The string whose length you want to calculate.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strlen()* function computes the length of the string pointed to by *s*.

Returns:

The number of characters that precede the terminating null character.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%d\n", strlen( "Howdy" ) );
    printf( "%d\n", strlen( "Hello world\n" ) );
    printf( "%d\n", strlen( "" ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
5
12
0
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strlwr()

Convert a string to lowercase

Synopsis:

```
#include <string.h>

char* strlwr( char* s1 );
```

Arguments:

s1

The string that you want to convert to lowercase.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *strlwr()* function replaces the string *s1* with lowercase characters, by invoking [tolower\(\)](#) (p. 3482) for each character in the string.

Returns:

The address of the string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A mixed-case STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strlwr( source ) );
    printf( "%s\n", source );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A mixed-case STRING
a mixed-case string
A mixed-case string
```

Classification:*Unix* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strncasecmp()

Compare two strings, ignoring case, up to a given length

Synopsis:

```
#include <strings.h>

int strncasecmp( const char* str1,
                 const char* str2,
                 size_t n );
```

Arguments:

str1, str2

The strings that you want to compare.

n

The maximum number of characters that you want to compare.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *strncasecmp()* function compares up to *n* characters in two strings, specified by *str1* and *str2*, ignoring the case of the characters.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

The following code:

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>

void compare( const char *s1, const char *s2 )
{
    int retval;

    retval = strncasecmp( s1, s2, 3 );
    if( retval > 0 ) {
        printf( "%s > %s\n", s1, s2 );
    } else if( retval < 0 ) {
        printf( "%s < %s\n", s1, s2 );
    } else {
        printf( "%s == %s\n", s1, s2 );
    }
}

int main( void )
{
    char *str1 = "abcdefg";
    char *str2 = "HIJ";
    char *str3 = "Abc";
    char *str4 = "aBCDEfg";

    compare( str1, str2 );
    compare( str1, str3 );
    compare( str1, str4 );
    compare( str1, str1 );

    compare( str2, str2 );
    compare( str2, str3 );
    compare( str2, str4 );
    compare( str2, str1 );

    return EXIT_SUCCESS;
}
```

produces output that looks like:

```
abcdefg < HIJ
abcdefg == Abc
abcdefg == aBCDEfg
abcdefg == abcdefg
HIJ == HIJ
HIJ > Abc
HIJ > aBCDEfg
HIJ > abcdefg
```

Classification:

[*POSIX 1003.1 XSI*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strncat()

Concatenate two strings, up to a maximum length

Synopsis:

```
#include <string.h>

char* strncat( char* dst,
               const char* src,
               size_t n );
```

Arguments:

dst, src

The strings that you want to concatenate.

n

The maximum number of characters that you want to add from the *src* string.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strncat()* function appends no more than *n* characters of the string pointed to by *src* to the end of the string pointed to by *dst*. The first character of *src* overwrites the null character at the end of *dst*. This function always adds a terminating null character to the result.

Returns:

The same pointer as *dst*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char buffer[80];

int main( void )
{
    strcpy( buffer, "Hello " );
    strncat( buffer, "world", 8 );
```

```
printf( "%s\n", buffer );
strncat( buffer, "*****", 4 );
printf( "%s\n", buffer );
return EXIT_SUCCESS;
}
```

produces the output:

```
Hello world
Hello world****
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strncmp()

Compare two strings, up to a given length

Synopsis:

```
#include <string.h>

int strncmp( const char* s1,
             const char* s2,
             size_t n );
```

Arguments:

s1, s2

The strings that you want to compare.

n

The maximum number of characters that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strncmp()* function compares up to *n* characters from the strings pointed to by *s1* and *s2*.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", strncmp( "abcdef", "abcDEF", 10 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF", 6 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF", 3 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF", 0 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
1
1
0
0
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strncpy()

Copy a string, to a maximum length

Synopsis:

```
#include <string.h>

char* strncpy( char* dst,
               const char* src,
               size_t n );
```

Arguments:

dst

A pointer to where you want to copy the string.

src

The string that you want to copy.

n

The maximum number of characters that you want to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strncpy()* function copies no more than *n* characters from the string pointed to by *src* into the array pointed to by *dst*.



Copying of overlapping objects isn't guaranteed to work properly. See the [memmove\(\)](#) (p. 1618) function if you wish to copy objects that overlap.

If the string pointed to by *src* is shorter than *n* characters, null characters are appended to the copy in the array pointed to by *dst*, until *n* characters in all have been written. If the string pointed to by *src* is longer than *n* characters, then the result isn't terminated by a null character.

Returns:

The same pointer as *dst*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char buffer[15];

    printf( "%s\n", strncpy( buffer, "abcdefg", 10 ) );
    printf( "%s\n", strncpy( buffer, "1234567", 6 ) );
    printf( "%s\n", strncpy( buffer, "abcdefg", 3 ) );
    printf( "%s\n", strncpy( buffer, "*****", 0 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
abcdefg
123456g
abc456g
abc456g
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strnicmp()

Compare two strings up to a given length, ignoring case

Synopsis:

```
#include <string.h>

int strnicmp( const char* s1,
              const char* s2,
              size_t len );
```

Arguments:

s1, s2

The strings that you want to compare.

len

The maximum number of characters that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strnicmp()* function compares up to *len* characters from the strings pointed to by *s1* and *s2*, ignoring case.

Returns:

< 0

s1 is less than *s2*.

0

s1 is equal to *s2*.

> 0

s1 is greater than *s2*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 10 ) );
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 6 ) );
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 3 ) );
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 0 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
-20
-20
0
0
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strnlen()

Compute the length of a string, to a maximum number of bytes

Synopsis:

```
#include <string.h>

size_t strnlen( const char * s,
                size_t maxlen );
```

Arguments:

s

The string whose length you want to calculate.

maxlen

The maximum length to check.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *strnlen()* function computes the length of the string pointed to by *s*, not including the terminating null character, up to a maximum of *maxlen* bytes. The function doesn't check any more than the first *maxlen* bytes.

Returns:

The minimum of *maxlen* and the number of characters that precede the terminating null character.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%d\n", strnlen( "Howdy", 10 ) );
    printf( "%d\n", strnlen( "Hello world\n", 5 ) );
    printf( "%d\n", strnlen( "", 10 ) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
5  
5  
0
```

Classification:

POSIX 1003.1-2008 CX (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strnset()

Fill a string with a given character, to a given length

Synopsis:

```
#include <string.h>

char * strnset( char * s1,
                int fill,
                size_t len );
```

Arguments:

s1

The string that you want to fill.

fill

The value that you want to fill the string with.

len

The number of characters to fill.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strnset()* function fills the string *s1* with the value of the argument *fill*, converted to be a character value. When the value of *len* is greater than the length of the string, the entire string is filled. Otherwise, that number of characters at the start of the string are set to the fill character.

Returns:

The address of the string, *s1*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
char source[] = { "A sample STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strnset( source, '=', 100 ) );
    printf( "%s\n", strnset( source, '*', 7 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
=====
*****=====
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strpbrk()

Find the first character in a string that's in a given character set

Synopsis:

```
#include <string.h>

char* strpbrk(char* str,
              char* charset );
```

Arguments:

str

The string that you want to search.

charset

The set of characters you want to look for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strpbrk()* function locates the first occurrence in the string pointed to by *str* of any character from the string pointed to by *charset*.

Returns:

A pointer to the located character, or `NULL` if no character from *charset* occurs in *str*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char* p = "Find all vowels";

    while( p != NULL ) {
        printf( "%s\n", p );
        p = strpbrk( p+1, "aeiouAEIOU" );
    }
    return EXIT_SUCCESS;
}
```

produces the output:

```
Find all vowels  
ind all vowels  
all vowels  
owels  
els
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strptime()

Convert a string into a time

Synopsis:

```
#include <time.h>

char * strptime( const char *buf,
                 const char *format,
                 struct tm *timeptr );
```

Arguments:

buf

A pointer to a buffer that contains the string to convert.

format

The format that you want to use for the time; see “[Formats](#) (p. 3233),” below.

timeptr

A pointer to a [tm](#) (p. 3476) structure where the function can store the time.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *strptime()* function converts the character string that *buf* points to into a time and stores the time in the `tm` structure pointed to by *timeptr*, using the specified format.

Formats

The *format* is composed of zero or more directives, each of which is composed of one of the following:

- one or more whitespace characters (as specified by [isspace\(\)](#) (p. 1451))
- an ordinary character (neither % nor a whitespace character)

- a conversion specification, composed of a % character followed by a conversion character that specifies the replacement required.



Ensure that there are whitespace or other non-alphanumeric characters between any two conversion specifications.

The *strptime()* function supports the following conversion specifications:

%a or %A

The day of the week, using the locale's weekday names; you can use either the abbreviated or full name.

%b, %B, or %h

The month, using the locale's month names; you can use either the abbreviated or full name.

%c

Replaced by the locale's appropriate date and time representation.

%C

The century number [00,99]. Leading zeros are permitted but not required.

%d or %e

The day of the month [01,31]. Leading zeros are permitted but not required.

%D

The date as %m/%d/%y.

%H

The hour (24-hour clock) [00,23]; leading zeros are permitted but not required.

%I

The hour (12-hour clock) [01,12]; leading zeros are permitted but not required.

%j

The day number of the year [001,366]; leading zeros are permitted but not required.

%m

The month number [01,12]; leading zeros are permitted but not required.

%M

The minute [00,59]; leading zeros are permitted but not required.

%n or %t

Any whitespace.

%p

The locale's equivalent of a.m. or p.m.

%r

12-hour clock time using the AM/PM notation if *t_fmt_ampm* isn't an empty string in the *LC_TIME* portion of the current locale; in the POSIX locale, this is equivalent to %I:%M:%S %p.

%R

The time as %H:%M.

%S

The seconds [00,60]; leading zeros are permitted but not required.

%T

The time as %H:%M:%S.

%U

The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.

%w

The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.

%W

The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.

%x

The date, using the locale's date format.

%X

The time, using the locale's time format.

%y

The year within the century. When a century isn't otherwise specified, values in the range [69,99] refer to years 1969 to 1999 inclusive, and values in

the range [00,68] refer to years 2000 to 2068 inclusive; leading zeros are permitted but not required.



The default century inferred from a 2-digit year will likely change in a future version of IEEE Std 1003.1-2001. (This would apply to all commands accepting a 2-digit year as input.)

%Y

The year, including the century (for example, 1988).

%%

Replaced by %.

Modified conversion specifiers

You can modify some conversion specifiers by adding the E and O (“Oh”) modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification doesn't exist in the current locale, the function behaves as if you used the unmodified conversion specification.

%Ec

The locale's alternative appropriate date and time representation.

%EC

The name of the base year (period) in the locale's alternative representation.

%Ex

The locale's alternative date representation.

%EX

The locale's alternative time representation.

%Ey

The offset from %EC (year only) in the locale's alternative representation.

%EY

The full alternative year representation.

%Od or %Oe

The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.

%OH

The hour (24-hour clock) using the locale's alternative numeric symbols.

%OI

The hour (12-hour clock) using the locale's alternative numeric symbols.

%Om

The month using the locale's alternative numeric symbols.

%OM

The minutes using the locale's alternative numeric symbols.

%OS

The seconds using the locale's alternative numeric symbols.

%OU

The week number of the year (using Sunday as the first day of the week) using the locale's alternative numeric symbols.

%Ow

The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW

The week number of the year (using Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy

The year (offset from %C) using the locale's alternative numeric symbols.

A conversion specification composed of whitespace characters is executed by scanning input up to the first character that isn't whitespace (which remains unscanned), or until no more characters can be scanned.

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of conversion specifications composed of %n, %t, whitespace characters, or any combination is executed by scanning up to the first character that isn't whitespace (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned.

These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate tm structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf*, such as month or weekday names. If no match is found, *strptime()* fails and doesn't scan any more characters.

Returns:

A pointer to the character after the last character parsed in the string, or NULL.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

int main( void )
{
    struct tm my_tm;
    char in_buffer[ 80 ] ="July 31, 1993 11:00:00",
        out_buffer[ 80 ];
    time_t t;

    /* Convert the string to a struct tm. */
    memset (&my_tm, 0, sizeof(struct tm));
    strptime( in_buffer, "%b %d, %Y %T", &my_tm );

    /* Convert the struct tm to a time_t (to fill in the
     * missing fields). */
    t = mktime (&my_tm);

    /* Convert the time back to a string. */
    strftime( out_buffer, 80, "That's %D (a %A), at %T",
              localtime (&t) );
    printf( "%s\n", out_buffer );

    return EXIT_SUCCESS;
}
```

This produces the output:

That's 07/31/93 (a Saturday), at 11:00:00

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

strchr()

Find the last occurrence of a character in a string

Synopsis:

```
#include <string.h>

const char* strchr(const char* s,
                   int c );
```

Arguments:

s

The string that you want to search.

c

The character that you're looking for.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *strchr()* function locates the last occurrence of *c* (converted to a char) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

Returns:

A pointer to the located character, or a NULL pointer if the character doesn't occur in the string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%s\n", strchr( "abcdeabcde", 'a' ) );
    if( strchr( "abcdeabcde", 'x' ) == NULL )
        printf( "NULL\n" );
    return EXIT_SUCCESS;
}
```

produces the output:

```
abcde  
NULL
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strrev()

Reverse a string

Synopsis:

```
#include <string.h>

char* strrev( char* s1 );
```

Arguments:

s1

The string that you want to reverse.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strrev()* function replaces the string *s1* with a string whose characters are in the reverse order.

Returns:

The address of the string, *s1*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A sample STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strrev( source ) );
    printf( "%s\n", strrev( source ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
GNIRTS elpmas A
A sample STRING
```

Classification:[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strsep()

Separate a string into pieces marked by given delimiters

Synopsis:

```
#include <string.h>

char *strsep( char **stringp,
              char *delim );
```

Arguments:***stringp***

The address of a pointer to the string that you want to break into pieces; see below.

delim

A set of characters that delimit the pieces in the string.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *strsep()* function looks in the null-terminated string pointed to by *stringp* for the first occurrence of any character in *delim* and replaces this with a \0, records the location of the next character in **stringp*, then returns the original value of **stringp*. If no delimiter characters are found, *strsep()* sets **stringp* to NULL; if **stringp* is initially NULL, *strsep()* returns NULL.

Returns:

A pointer to the original value of **stringp*.

Examples:

Parse strings containing runs of whitespace, making up an argument vector:

```
char inputstring[100];
char **argv[51], **ap = argv, *p, *val;

/* set up inputstring */
for (p = inputstring; p != NULL; ) {
```

```
    while ((val = strsep(&p, " \t")) != NULL && *val == '\0');  
    *ap++ = val;  
}  
*ap = 0;
```

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strset()

Fill a string with a given character

Synopsis:

```
#include <string.h>

char* strset( char* s1,
              int fill );
```

Arguments:

s1

The string that you want to fill.

fill

The value that you want to fill the string with.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strset()* function fills the string pointed to by *s1* with the character *fill*. The terminating null character in the original string remains unchanged.

Returns:

The address of the string, *s1*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A sample STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strset( source, '=' ) );
    printf( "%s\n", strset( source, '*' ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
=====
*****
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strsignal()

Return the description of a signal

Synopsis:

```
#include <string.h>
char *strsignal( int signo );
```

Arguments:

signo

The signal number that you want the description of.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strsignal()* function returns a pointer to the language-dependent string describing a signal.

Returns:

A pointer to the description of the signal, or `NULL` if *signo* isn't a valid signal number. This array will be overwritten by subsequent calls to *strsignal()*.



Don't modify the array returned by this function.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

strspn()

Count the characters at the beginning of a string that are in a given character set

Synopsis:

```
#include <string.h>

size_t strspn( const char* str,
               const char* charset );
```

Arguments:

str

The string that you want to search.

charset

The set of characters you want to look for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strspn()* function computes the length of the initial segment of the string pointed to by *str* that consists of characters from the string pointed to by *charset*. The terminating null character isn't considered to be part of *charset*.

Returns:

The length of the segment.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%d\n", strspn( "out to lunch", "aeiou" ) );
    printf( "%d\n", strspn( "out to lunch", "xyz" ) );
    return EXIT_SUCCESS;
}
```

produces the output:

2
0

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strstr()

Find one string inside another

Synopsis:

```
#include <string.h>

char* strstr(char* str,
             char* substr );
```

Arguments:

str

The string that you want to search.

substr

The string that you're looking for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strstr()* function locates the first occurrence in the string pointed to by *str* of the sequence of characters (excluding the terminating null character) in the string pointed to by *substr*.

Returns:

A pointer to the located string, or `NULL` if the string isn't found.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%s\n", strstr("This is an example", "is") );
    return EXIT_SUCCESS;
}
```

produces the output:

is is an example

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strtod(), strtof(), strtold()*Convert a string into a double-precision number***Synopsis:**

```
#include <stdlib.h>

double strtod( const char *ptr,
               char **endptr );

float strtof( const char *ptr,
              char **endptr );

long double strtold( const char *ptr,
                    char **endptr );
```

Arguments:***ptr***

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

Library:**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strtod()*, *strtof()*, and *strtold()* functions convert the string pointed to by *ptr* into a double-precision representation:

This function:	Returns:
<i>strtod()</i>	double
<i>strtof()</i>	float
<i>strtold()</i>	long double

These functions skip any leading white space, and then look for a subject sequence that consists of an optional plus or minus sign followed by one of the following:

- a non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- INF or INFINITY, ignoring case
- NAN, ignoring case



POSIX says that these functions can optionally parse an “n-char sequence” after the NAN. The QNX Neutrino versions don't interpret an n-char sequence, and the result is equivalent to specifying NAN without the n-char sequence.

The conversion ends at the first unrecognized character. If *endptr* isn't NULL, a pointer to the unrecognized character is stored in the object *endptr* points to.

Returns:

The converted value. If the correct value would cause overflow, plus or minus HUGE_VAL is returned according to the sign, and *errno* (p. 539) is set to ERANGE. If the correct value would cause underflow, then zero is returned, and *errno* is set to ERANGE.



These functions return zero and set *errno* if the input string can't be converted; they don't change *errno* if no errors occurred. If you want to check for errors, set *errno* (p. 539) to 0, call the function, and then check *errno* again.

Errors:

ERANGE

The value to be returned would cause overflow or underflow.

EINVAL

No conversion could be performed.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    double pi;

    pi = strtod( "3.141592653589793", NULL );
    printf( "pi=%17.15f\n", pi );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

strtoimax(), strtoumax()

Convert a string into an integer

Synopsis:

```
#include <inttypes.h>

intmax_t strtoimax ( const char * nptr,
                    char ** endptr,
                    int base );

uintmax_t strtoumax ( const char * nptr,
                     char ** endptr,
                     int base );
```

Arguments:

nptr

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

base

The base of the number being parsed:

- If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
- If *base* isn't zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strtoimax()` and `strtoumax()` functions are the same as the [strtol\(\)](#), [strtoll\(\)](#) (p. 3262), [strtoul\(\)](#), and [strtoull\(\)](#) (p. 3264) functions except that they return objects of type `intmax_t` and `uintmax_t`.

Returns:

The converted value.

If the correct value causes an overflow, `INTMAX_MAX|UINTMAX_MAX` or `INTMAX_MIN` is returned according to the sign and [errno](#) (p. 539) is set to `ERANGE`. If `base` is out of range, zero is returned and `errno` is set to `EINVAL`.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strtok()

Break a string into tokens

Synopsis:

```
#include <string.h>

char* strtok( char* s1,
              const char* s2 );
```

Arguments:

s1

NULL, or the string that you want to break into tokens; see below.

s2

A set of the characters that separate the tokens.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strtok()* function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*.

The first call to *strtok()* returns a pointer to the first token in the string pointed to by *s1*. Subsequent calls to *strtok()* must pass a NULL pointer as the first argument, in order to get the next token in the string. The set of delimiters used in each of these calls to *strtok()* can be different from one call to the next.

The first call in the sequence searches *s1* for the first character that isn't contained in the current delimiter string *s2*. If no such character is found, then there are no tokens in *s1*, and *strtok()* returns a NULL pointer. If such a character is found, it's the start of the first token.

The *strtok()* function then searches from there for a character that's contained in the current delimiter string. If no such character is found, the current token extends to the end of the string pointed to by *s1*. If such a character is found, it's overwritten by a null character, which terminates the current token. The *strtok()* function saves a

pointer to the following character, from which the next search for a token will start when the first argument is a **NULL** pointer.



You might want to keep a copy of the original string because *strtok()* is likely to modify it.

Returns:

A pointer to the token found, or **NULL** if no token was found.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char* p;
    char* buffer;
    char* delims = { " .," };

    buffer = strdup( "Find words, all of them." );
    printf( "%s\n", buffer );
    p = strtok( buffer, delims );
    while( p != NULL ) {
        printf( "word: %s\n", p );
        p = strtok( NULL, delims );
    }
    printf( "%s\n", buffer );
    return EXIT_SUCCESS;
}
```

produces the output:

```
Find words, all of them.
word: Find
word: words
word: all
word: of
word: them
Find
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

strtok_r()

Break a string into tokens (reentrant)

Synopsis:

```
#include <string.h>

char* strtok_r( char* s,
                const char* sep,
                char** lasts );
```

Arguments:

s1

NULL, or the string that you want to break into tokens; see below.

s2

A set of the characters that separate the tokens.

lasts

The address of a pointer to a character, which the function can use to store information necessary for it to continue scanning the same string.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The function *strtok_r()* breaks the string *s* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *sep*.

In the first call to *strtok_r()*, *s* must point to a null-terminated string, *sep* points to a null-terminated string of separator characters, and *lasts* is ignored. The *strtok_r()* function returns a pointer to the first character of the first token, writes a NULL character into *s* immediately following the returned token, and updates *lasts*.

In subsequent calls, *s* must be a NULL pointer and *lasts* must be unchanged from the previous call so that subsequent calls will move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no tokens remain in *s*, a NULL pointer is returned.



You might want to keep a copy of the original string because *strtok_r()* is likely to modify it.

Returns:

A pointer to the token found, or NULL if no token was found.

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strtol(), strtoll()

Convert a string into a long integer

Synopsis:

```
#include <stdlib.h>

long int strtol( const char * ptr,
                char ** endptr,
                int base );

long long strtoll( const char * ptr,
                  char ** endptr,
                  int base );
```

Arguments:

ptr

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

base

The base of the number being parsed:

- If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
- If *base* isn't zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *strtol()* function converts the string pointed to by *ptr* to an object of type `long int`; *strtoll()* converts the string pointed to by *ptr* to an object of type `long long`.

These functions recognize strings that contain the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits and letters.

The conversion ends at the first unrecognized character. If *endptr* isn't `NULL`, a pointer to the unrecognized character is stored in the object *endptr* points to.

Returns:

The converted value.

If the correct value causes an overflow, the function will return either `LONG_MAX` or `LONG_MIN` depending to the sign, and *errno* will be set to `ERANGE`. If *base* is out of range, the function returns zero and sets *errno* to `EINVAL`.

Examples:

```
#include <stdlib.h>

int main( void )
{
    long int v;

    v = strtol( "12345678", NULL, 10 );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strtoul(), strtoull()

Convert a string into an unsigned long integer

Synopsis:

```
#include <stdlib.h>

unsigned long int strtoul( const char * ptr,
                        char ** endptr,
                        int base );

unsigned long long strtoull( const char * ptr,
                          char ** endptr,
                          int base );
```

Arguments:

ptr

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

base

The base of the number being parsed:

- If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
- If *base* isn't zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *strtol()* function converts the string pointed to by *ptr* to an unsigned long; *strtoull()* converts the string pointed to by *ptr* to an unsigned long long.

These functions recognize strings that contain the following:

- optional white space
- a sequence of digits and letters.

The conversion ends at the first unrecognized character. A pointer to that character is stored in the object *endptr* points to, if *endptr* isn't NULL.

Returns:

The converted value.

If the correct value causes an overflow, the function will return `ULONG_MAX` or `ULLONG_MAX` and set *errno* to `ERANGE`. If *base* is out of range, the function returns zero and sets *errno* to `EINVAL`.

Examples:

```
#include <stdlib.h>

int main( void )
{
    unsigned long int v;

    v = strtoul( "12345678", NULL, 10 );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strupr()

Convert a string to uppercase

Synopsis:

```
#include <string.h>

char* strupr( char* s1 );
```

Arguments:

s1

The string that you want to convert to uppercase.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strupr()* function replaces the string *s1* with uppercase characters, by invoking [toupper\(\)](#) (p. 3484) for each character in the string.

Returns:

The address of the string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A mixed-case STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strupr( source ) );
    printf( "%s\n", source );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A mixed-case STRING
A MIXED-CASE STRING
A MIXED-CASE STRING
```

Classification:[Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

strxfrm()

Transform one string into another, to a given length

Synopsis:

```
#include <string.h>

size_t strxfrm( char* dst,
                const char* src,
                size_t n );
```

Arguments:

dst

The string that you want to transform.

src

The string that you want to place in *dst*.

n

The maximum number of characters to transform.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *strxfrm()* function transforms, for no more than *n* characters, the string pointed to by *src* to the buffer pointed to by *dst*. The transformation uses the collating sequence selected by [setlocale\(\)](#) (p. 2880) so that two transformed strings compare identically (using the [strncmp\(\)](#) (p. 3221) function) to a comparison of the original two strings using [strcoll\(\)](#) (p. 3190).

If the collating sequence is selected from the "C" locale, *strxfrm()* is equivalent to [strncpy\(\)](#) (p. 3223), except that *strxfrm()* doesn't pad the *dst* argument with null characters when the argument *src* is shorter than *n* characters.

Returns:

The length of the transformed string. If this length is more than *n*, the contents of the array pointed to by *dst* are indeterminate.



If an error occurs, *strxfrm()* sets *errno* (p. 539) and returns 0. Since the function could also return zero on success, the only way to tell that an error has occurred is to set *errno* to 0 before calling *strxfrm()* and check *errno* afterward.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

char src[] = { "A sample STRING" };
char dst[20];

int main( void )
{
    size_t len;

    setlocale( LC_ALL, "C" );
    printf( "%s\n", src );
    len = strxfrm( dst, src, 20 );
    printf( "%s (%u)\n", dst, len );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
A sample STRING (15)
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

swab()

Endian-swap a given number of bytes

Synopsis:

```
#include <unistd.h>

void swab( const void * src,
           void * dest,
           ssize_t nbytes );
```

Arguments:

- src**
A pointer to the buffer that you want to copy the bytes from.
- dest**
A pointer to the buffer where you want the function to copy the bytes.
- nbytes**
The number of bytes that you want to copy and swap.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *swab()* function copies *nbytes* bytes, pointed to by *src*, to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even.

If <i>nbytes</i> is:	Then:
Odd	<i>nbytes</i> -1 bytes are copied and exchanged. The disposition of the last byte is unspecified.
Negative	<i>swab()</i> does nothing.

If copying takes place between objects that overlap, the behavior is undefined.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

swprintf()

Print formatted wide-character output into a string

Synopsis:

```
#include <wchar.h>

int swprintf( wchar_t * ws,
              size_t n,
              const wchar_t * format,
              ... );
```

Arguments:

ws

A pointer to the buffer where you want the function to store the formatted string.

n

The maximum number of wide characters to store in the buffer, including a terminating null character.

format

A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *swprintf()* function is similar to [fwprintf\(\)](#) (p. 806) except that *swprintf()* places the generated output into the wide-character array pointed to by *buf*, instead of writing it to a file. A null character is placed at the end of the generated character string.

The *swprintf()* function is the wide-character version of [sprintf\(\)](#) (p. 3151).

Returns:

The number of wide characters written, excluding the terminating NUL, or a negative number if an error occurred ([errno](#) (p. 539) is set).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call this function in a signal handler if the data isn't floating point.

swscanf()

Scan input from a wide-character string

Synopsis:

```
#include <wchar.h>

int swscanf( const wchar_t * ws,
             const wchar_t * format,
             ... );
```

Arguments:

ws

The wide-character string that you want to read from.

format

A wide-character string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *swscanf()* function scans input from the wide-character string *ws*, under control of the argument *format*. Following the format string is the list of addresses of items to receive values.

The *swscanf()* function is the wide-character version of [sscanf\(\)](#) (p. 3162).

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is terminated by reaching the end of the input string.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call this function in a signal handler if the data isn't floating point.

symlink()

Create a symbolic link to a path

Synopsis:

```
#include <unistd.h>

int symlink( const char* pname,
             const char* slink );
```

Arguments:

pname

The path that you want to link to.

slink

The symbolic link that you want to create.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *symlink()* function creates a symbolic link named *slink* that contains the pathname specified by *pname* (*slink* is the name of the symbolic link created, *pname* is the pathname contained in the symbolic link).

File access checking isn't performed on the file named by *pname*, and the file need not exist.

If the *symlink()* function is unsuccessful, any file named by *slink* is unaffected.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EACCES**

A component of the *slink* path prefix denies search permission, or write permission is denied in the parent directory of the symbolic link to be created.

EEXIST

A file named by *slink* already exists.

ELOOP

A loop exists in symbolic links encountered during resolution of the *slink* argument, and it resolves to more than SYMLINK_MAX levels.

ENAMETOOLONG

A component of the path specified by *slink* exceeds NAME_MAX bytes, or the length of the entire pathname exceeded PATH_MAX characters.

ENOSPC

The new symbolic link can't be created because there's no space left on the filesystem that will contain the symbolic link.

ENOSYS

The *symlink()* function isn't implemented for the filesystem specified in *slink*.

ENOTDIR

A component of the path prefix of *slink* isn't a directory.

EROFS

The file *slink* would reside on a read-only filesystem.

Examples:

```
/*
 * create a symbolic link to "/usr/nto/include"
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    if( symlink( "/usr/nto/include", "slink" ) == -1 ) {
        perror( "slink -> /usr/nto/include" );
        exit( EXIT_FAILURE );
    }
    exit( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sync()

Synchronize filesystem updates

Synopsis:

```
#include <unistd.h>

void sync( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sync()` function queues all the modified block buffers for writing, and returns; it doesn't wait for the actual I/O to take place. Use this function — or [fsync\(\)](#) (p. 787) for a single file — to ensure consistency of the entire on-disk filesystem with the contents of the in-memory buffer cache.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncCondvarSignal(), SyncCondvarSignal_r()

Wake up any threads that are blocked on a synchronization object

Synopsis:

```
#include <sys/neutrino.h>

int SyncCondvarSignal( sync_t* sync,
                      int broadcast );

int SyncCondvarSignal_r( sync_t* sync,
                        int broadcast );
```

Arguments:

sync

A pointer to a `sync_t` for the synchronization object. You must have initialized this argument by calling [SyncTypeCreate\(\)](#) (p. 3305) or statically initialized it with the manifest `PTHREAD_COND_INITIALIZER`.

broadcast

Zero if you want to make ready to run the thread with the highest priority that's been waiting the longest, or nonzero if you want to make all waiting threads ready.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncCondvarSignal()* and *SyncCondvarSignal_r()* kernel calls wake up one or all threads that are blocked on the synchronization object *sync*.

These functions are similar, except in the way they indicate errors. See the [Returns](#) (p. 3281) section for details.



Instead of using these kernel calls directly, consider calling [pthread_cond_broadcast\(\)](#) (p. 2257) or [pthread_cond_signal\(\)](#) (p. 2263).

In all cases, each awakened thread attempts to reacquire the controlling mutex passed in [SyncCondvarWait\(\)](#) (p. 3282) before control is returned to the thread. If the mutex is

already locked when the kernel attempts to lock it, the thread becomes blocked on the mutex until it's unlocked.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SyncCondvarSignal()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncCondvarSignal_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EFAULT

A fault occurred when the kernel tried to access *sync*.

EINVAL

The synchronization ID specified in *sync* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncCondvarWait(), SyncCondvarWait_r()

Block a thread on a synchronization object

Synopsis:

```
#include <sys/neutrino.h>

int SyncCondvarWait( sync_t * sync,
                    sync_t * mutex );

int SyncCondvarWait_r( sync_t * sync,
                      sync_t * mutex );
```

Arguments:

sync

A pointer to a `sync_t` for the synchronization object. You must have initialized this argument by calling [SyncTypeCreate\(\)](#) (p. 3305) or statically initialized it with the manifest `PTHREAD_COND_INITIALIZER`.

mutex

The mutex that's associated with the condition variable. You must lock this mutex by calling [SyncMutexLock\(\)](#) (p. 3294) (or the POSIX [pthread_mutex_lock\(\)](#) (p. 2318) cover routine). The kernel releases the mutex lock in the kernel when it blocks the thread on *sync*.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncCondvarWait()* and *SyncCondvarWait_r()* kernel calls block the calling thread on the synchronization object, *sync*. If more than one thread is blocked on the object, they're queued in priority order.

These functions are similar, except in the way they indicate errors. See the [Returns](#) (p. 3284) section for details.



Instead of using these kernel calls directly, consider calling [pthread_cond_timedwait\(\)](#) (p. 2265) or [pthread_cond_wait\(\)](#) (p. 2269).

The blocked thread can be unblocked by any one of the following conditions:

Condition variable signalled

The condition variable was signaled by a call to [SyncCondvarSignal\(\)](#) (p. 3280), that determined that this thread should be awakened.

Before returning from *SyncCondvarWait()*, *mutex* is reacquired. If *mutex* is locked, the thread enters into the STATE_MUTEX state waiting for *mutex* to be unlocked. At this point it's as though you had called *SyncMutexLock(mutex)*.

Timeout

The wait was terminated by a timeout initiated by a previous call to [TimerTimeout\(\)](#) (p. 3464).

Before returning from *SyncCondvarWait()*, *mutex* is reacquired. If *mutex* is locked, the thread enters into the STATE_MUTEX state waiting for *mutex* to be unlocked. At this point it's as though you had called *SyncMutexLock(mutex)*.

POSIX signal

The wait was terminated by an unmasked signal initiated by a call to [SignalKill\(\)](#) (p. 2986). If a signal handler has been set up, the signal handler runs with *mutex* unlocked. On return from the signal handler, *mutex* is reacquired. If *mutex* is locked, the thread enters into the STATE_MUTEX state waiting for *mutex* to be unlocked. At this point, it's as though you had called *SyncMutexLock(mutex)*.

Thread cancellation

The wait was terminated by a thread cancellation initiated by a call to [ThreadCancel\(\)](#) (p. 3399). Before calling the cancellation handler, *mutex* is reacquired. If *mutex* is locked, the thread enters into the STATE_MUTEX state waiting for *mutex* to be unlocked. At this point, it's as though you had called *SyncMutexLock(mutex)*.

In all cases, *mutex* is reacquired before the call returns. If the thread enters the STATE_MUTEX state, the rules governing *SyncMutexLock()* are in effect.

Condition variables are used to block a thread until a certain condition is satisfied. Spurious wakeups may occur due to timeouts, signals, and broadcast condition variable signals. Therefore, you should always reevaluate the condition, even on a successful return. The easiest way to do this is with a while loop. For example:

```
SyncMutexLock(&mutex);
while(some_condition) {
    SyncCondvarWait(&condvar, &mutex);
}
```

```
SyncMutexUnlock(&mutex);
```

Blocking states

STATE_CONDVAR

The calling thread blocks waiting for the condition variable to be signaled.

STATE_MUTEX

The thread was unblocked from the STATE_CONDVAR state and while trying to reacquire the controlling mutex, found the mutex was locked by another thread.

Returns:

The only difference between these functions is the way they indicate errors:

SyncCondvarWait()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncCondvarWait_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function returns any value in the Errors section.

Errors:

EAGAIN

On the first use of a statically initialized *sync*, all kernel synchronization objects were in use.

EFAULT

A fault occurred when the kernel tried to access *sync* or *mutex*.

EINVAL

The synchronization ID specified in *sync* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncCtl(), SyncCtl_r()

Perform an operation on a synchronization object

Don't use the *SyncCtl()* or *SyncCtl_r()* kernel call directly; instead, call one of the following:



Instead of this <i>cmd</i> :	Call:
_NTO_SCTL_GETPRIOCEILING	pthread_mutex_getprioceiling() (p. 2314)
_NTO_SCTL_SETPRIOCEILING	pthread_mutex_setprioceiling() (p. 2322)
_NTO_SCTL_SETEVENT	SyncMutexEvent() (p. 3291)
_NTO_SCTL_MUTEX_WAKEUP	pthread_mutex_wakeup_np() (p. 2331)

Synopsis:

```
#include <sys/neutrino.h>

int SyncCtl( int cmd,
             sync_t * sync,
             void * data );

int SyncCtl_r( int cmd,
               sync_t * sync,
               void * data );
```

Arguments:

cmd

The operation type; one of:

- *_NTO_SCTL_GETPRIOCEILING* — get the ceiling priority of the mutex pointed to by *sync* and put it in the variable pointed to by *data*.
- *_NTO_SCTL_SETPRIOCEILING* — return the original ceiling priority. Set the ceiling priority of the mutex pointed to by *sync* to the value pointed to by *data*.
- *_NTO_SCTL_SETEVENT* — attach an event, pointed to by *data*, to the mutex pointed to by *sync*.

- `_NTO_SCTL_MUTEX_WAKEUP` — wake up threads that are blocked on a mutex. The *data* argument points to a structure that specifies the process and thread IDs.

sync

A pointer to the synchronization object that you want to manipulate.

data

A pointer to data associated with the command, or a place where the function can store the requested information, depending on the operation.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncCtl()* and *SyncCtl_r()* kernel calls let you:

- set or get a ceiling priority for a mutex
- attach an event to a mutex so you'll be notified when the mutex changes to the DEAD state
- wake up threads that are blocked on a mutex

These functions are similar, except for the way they indicate errors. See the [Returns](#) (p. 3287) section for details.



In order to change the priority ceiling to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The only difference between these functions is the way they indicate errors:

SyncCtl()

If an error occurs, the function returns `-1` and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncCtl_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function returns any value listed in the Errors section.

Errors:**EAGAIN**

All kernel synchronization event objects are in use.

EFAULT

A fault occurred when the kernel tried to access *sync* or *data*.

EINVAL

The synchronization object pointed to by *sync* doesn't exist, or the ceiling priority value pointed to by *data* is out of range.

ENOSYS

The *SyncCtl()* and *SyncCtl_r()* functions aren't currently supported.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncDestroy(), SyncDestroy_r()

Destroy a synchronization object



Don't use the `SyncDestroy()` or `SyncDestroy_r()` kernel call directly; instead, use the POSIX functions for synchronization objects (see [pthread_cond_destroy\(\)](#) (p. 2259), [pthread_mutex_destroy\(\)](#) (p. 2312), [pthread_rwlock_destroy\(\)](#) (p. 2364), and [sem_destroy\(\)](#) (p. 2807)).

Synopsis:

```
#include <sys/neutrino.h>

int SyncDestroy( sync_t* sync );

int SyncDestroy_r ( sync_t* sync );
```

Arguments:

sync

The synchronization object that you want to destroy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncDestroy()` and `SyncDestroy_r()` kernel calls destroy a synchronization object previously allocated by a call to [SyncTypeCreate\(\)](#) (p. 3305). You can destroy a locked mutex only if you're its owner. If the object is a condition variable with waiting threads, the call fails. Any attempt to use `sync` after it is destroyed fails.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3289) section for details.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SyncDestroy()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncDestroy_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function can return any value listed in the Errors section.

Errors:**EBUSY**

The synchronization object is locked by a thread.

EFAULT

A fault occurred when the kernel tried to access *sync*.

EINVAL

The synchronization ID specified in *sync* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncMutexEvent(), SyncMutexEvent_r()

Attach an event to a mutex

Synopsis:

```
#include <sys/neutrino.h>

int SyncMutexEvent( sync_t * sync,
                   struct sigevent * event );

int SyncMutexEvent_r( sync_t * sync,
                     struct sigevent * event );
```

Arguments:

sync

A pointer to the synchronization object for the mutex that you want to attach an event to.

event

A pointer to the [sigevent](#) (p. 2961) structure that describes the event that you want to attach, or NULL if you want to detach the currently registered event.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncMutexEvent()* and *SyncMutexEvent_r()* kernel calls attach the specified *event* to a mutex pointed to by *sync*. This event is delivered when the mutex enters the DEAD state, which happens when a process holding the mutex removes its last mapping of the mutex without unlocking it (including when the process dies while holding the mutex in shared memory).



This is likely caused by a bug in the process. Although the OS provides a way to detect this situation and revive the mutex, the data that the mutex is protecting might be in an inconsistent state.

SyncMutexEvent() and *SyncMutexEvent_r()* are similar, except for the way they indicate errors. See the [Returns](#) (p. 3293) section for details.

When you're notified that a mutex has been put into the DEAD state, you can revive it with [SyncMutexRevive\(\)](#) (p. 3297).

If you call *SyncMutexEvent()* with a NULL event, the function deletes any existing event registration.



In order to use this function with a SIGEV_PULSE sigevent that's going to a channel that was created by a different process with a different user ID, your process must have the PROCMGR_AID_CONNECTION ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Managing the death of a mutex

Here's how you might use a mutex in shared memory between threads in two or more processes and attempt to deal with the death of one of the processes while it's holding the mutex.

1. Designate one process as the shared memory's owner/manager.
2. This process creates and initializes the mutex, and attaches an event using *SyncMutexEvent()*.
3. This process marks the shared memory area and the mutex as usable.
4. Everyone goes ahead and uses the mutex and shared memory area; everything goes well until a process dies.
5. If a process dies, and one of its threads holds the mutex (i.e., has locked it), then the kernel marks the mutex as DEAD and sends the event registered with *SyncMutexEvent()* to the shared memory's owner/manager process.

If it's the shared memory's owner/manager process that died, then there's no event delivered, and the mutex is marked as DESTROYED; you can use [procmgr_guardian\(\)](#) (p. 2172) to try to handle this case. If there's no event registered, the mutex is marked as DESTROYED. In both these cases, the kernel sends a SIGDEADLK signal to all threads that are waiting, without a timeout, on the mutex. If these threads have blocked SIGDEADLK, they're made READY, as are threads that are waiting on the mutex, but with a timeout.

6. It's up to the shared memory's owner/manager process to handle the event. It should have some way of telling other threads that the shared memory area controlled by the mutex, and the mutex itself, are currently unusable, and should also have a mechanism to indicate when they again become usable.
7. Other threads already waiting on the dead mutex continue to wait on it.
8. The process acting on the event calls *SyncMutexRevive()* on the dead mutex and revives and acquires the mutex in one atomic operation.

If you call [pthread_mutex_lock\(\)](#) (p. 2318):

- on a destroyed mutex, the function returns **EINVAL**
- on a dead mutex, your thread will be blocked until the mutex is revived and released, or the process that has the *SyncMutexEvent()* event registered dies, in which case the lock operation returns **EINVAL**

Returns:

The only difference between these functions is the way they indicate errors:

SyncMutexEvent()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncMutexEvent_r()

Returns **EOK** on success. This function does **NOT** set *errno*. If an error occurs, the function returns any value listed in the Errors section.

Errors:

EAGAIN

All kernel synchronization event objects are in use.

EFAULT

A fault occurred when the kernel tried to access *sync*.

EINVAL

The synchronization object pointed to by *sync* doesn't exist.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncMutexLock()*, *SyncMutexLock_r()

Lock a mutex synchronization object

Synopsis:

```
#include <sys/neutrino.h>

int SyncMutexLock( sync_t * sync );

int SyncMutexLock_r( sync_t * sync );
```

Arguments:

sync

A pointer to the synchronization object for the mutex that you want to lock.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncMutexLock()* and *SyncMutexLock_r()* kernel calls try to lock the mutex synchronization object *sync*. If the mutex isn't currently locked, the call returns immediately with the object locked. The mutex is considered unlocked if the owner field of *sync* is zero. Otherwise, the owner field of *sync* is treated as the thread ID of the current owner of the mutex.

These functions are similar, except for the way they indicate errors. See the [Returns](#) (p. 3295) section for details.



The POSIX functions [pthread_mutex_lock\(\)](#) (p. 2318) and [pthread_mutex_unlock\(\)](#) (p. 2329) are faster, since they can potentially avoid a kernel call.

If the mutex is already locked, the calling thread blocks on *sync* until it's unlocked by the owner. If more than one thread is blocked on *sync* they're queued in priority order.

If the priority of the blocking thread is higher than the thread that owns the mutex, the owner's priority is boosted to match that of the caller. In other words, the owner inherits the caller's priority if it's higher. If the owner's priority is boosted, it returns to its previous value before any boosts when the mutex is unlocked. Note that the owner may be boosted more than once as higher priority threads block on *sync*.

If a thread is boosted via this mechanism and subsequently changes its own priority, that priority takes immediate effect and also becomes the value it's returned to after it releases the mutex.

Waiting for a mutex isn't a cancellation point. If a signal is delivered to the thread while waiting for the mutex, the signal handler runs and, upon return from the handler, the thread resumes waiting for the mutex as if it wasn't interrupted.



Avoid timeouts on mutexes. Mutexes should be locked for brief periods of time, eliminating the need for a timeout.

The *sync* argument must have been initialized by a call to [SyncTypeCreate\(\)](#) (p. 3305) or have been statically initialized by the manifest PTHREAD_MUTEX_INITIALIZER.

Blocking states

STATE_MUTEX

The calling thread blocks waiting for the synchronization object to be unlocked.

Returns:

The only difference between these functions is the way they indicate errors:

SyncMutexLock()

If an error occurs, the function returns -1 and sets *errno* (p. 539). Any other value returned indicates success.

SyncMutexLock_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, this function returns any value listed in the Errors section.

Errors:

EAGAIN

On the first use of a statically initialized *sync*, all kernel synchronization objects were in use.

EFAULT

A fault occurred when the kernel tried to access the buffers you provided.

EINVAL

The synchronization ID specified in *sync* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncMutexRevive(), SyncMutexRevive_r()

Revive a mutex that's in the DEAD state

Synopsis:

```
#include <sys/neutrino.h>

int SyncMutexRevive( sync_t * sync );
int SyncMutexRevive_r( sync_t * sync );
```

Arguments:

sync

A pointer to the synchronization object for the mutex that you want to revive.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncMutexRevive()* and *SyncMutexRevive_r()* kernel calls revive a mutex, pointed to by *sync*, that's in the DEAD state. The mutex will be put into the LOCK state and will be owned by the calling thread. The mutex counter is set to one (for recursive mutexes).

These functions are similar, except for the way they indicate errors. See the [Returns](#) (p. 3297) section for details.

See [SyncMutexEvent\(\)](#) (p. 3291) for information on how to get notified when a mutex enters the DEAD state.

Returns:

The only difference between these functions is the way they indicate errors:

SyncMutexRevive()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncMutexRevive_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function returns any value listed in the Errors section.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers you provided.

EINVAL

The synchronization object pointed to by *sync* doesn't exist or wasn't in the DEAD state.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncMutexUnlock(), SyncMutexUnlock_r()

Unlock a mutex synchronization object

Synopsis:

```
#include <sys/neutrino.h>

int SyncMutexUnlock( sync_t * sync );
int SyncMutexUnlock_r( sync_t * sync );
```

Arguments:

sync

A pointer to the synchronization object for the mutex that you want to unlock.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncMutexUnlock()* and *SyncMutexUnlock_r()* kernel calls unlock the mutex passed as *sync*. If there are threads blocked on the mutex, the *owner* member of *sync* is set to the thread ID of the thread with the highest priority that has been waiting the longest and it's made ready to run. If no threads are waiting, it's set to zero.

These functions are similar, except for the way they indicate errors. See the [Returns](#) (p. 3299) section for details.

If the calling thread had its priority boosted while it owned the mutex, it returns to its normal priority.



The POSIX functions [pthread_mutex_lock\(\)](#) (p. 2318) and [pthread_mutex_unlock\(\)](#) (p. 2329) are faster, since they can potentially avoid a kernel call.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

SyncMutexUnlock()

If an error occurs, The function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncMutexUnlock_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function returns any value listed in the Errors section.

Errors:**EFAULT**

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The synchronization ID specified in *sync* doesn't exist. The calling thread doesn't own the mutex.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncSemPost(), SyncSemPost_r()

Increment a semaphore

Synopsis:

```
#include <sys/neutrino.h>

int SyncSemPost( sync_t* sync );
int SyncSemPost_r( sync_t* sync );
```

Arguments:

sync

A pointer to the synchronization object for the semaphore that you want to increment.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncSemPost()* and *SyncSemPost_r()* kernel calls increment the semaphore referenced by the *sync* argument. If any threads are blocked on the semaphore, the one waiting the longest is unblocked and allowed to run.

These functions are identical, except for the way they indicate errors. See the [Returns](#) (p. 3301) section for details.



You should use the POSIX [sem_post\(\)](#) (p. 2818) function instead of calling *SyncSemPost()* directly.

Returns:

The only difference between these functions is the way they indicate errors:

SyncSemPost()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

SyncSemPost_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function returns one of the values listed in the Errors section.

Errors:

EAGAIN

Not enough memory for the kernel to create the internal *sync* object.

EFAULT

Invalid pointer.

EINTR

A signal interrupted this function.

EINVAL

The *sync* argument doesn't refer to a valid semaphore.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncSemWait(), SyncSemWait_r()

Wait on a semaphore

Synopsis:

```
#include <sys/neutrino.h>

int SyncSemWait( sync_t* sync,
                int try );

int SyncSemWait_r( sync_t* sync,
                  int try );
```

Arguments:

sync

A pointer to the synchronization object for the semaphore that you want to wait on.

try

Nonzero if you want a conditional wait.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *SyncSemWait()* and *SyncSemWait_r()* kernel calls decrement the semaphore referred to by the *sync* argument. If the semaphore value isn't greater than zero and *try* is zero, then the calling process blocks until it can decrement the counter or the call is interrupted by signal.

These functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 3304) section for details.



Instead of using these kernel calls directly, consider calling [sem_timedwait\(\)](#) (p. 2820), [sem_trywait\(\)](#) (p. 2823) or [sem_wait\(\)](#) (p. 2827).

If *try* is nonzero, the function acts as a conditional wait. If the call would block, the semaphore is unmodified, and the call returns with an error.

Returns:

The only difference between these functions is the way they indicate errors:

SyncSemWait()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success (the semaphore was successfully decremented).

SyncSemWait_r()

Returns EOK on success (the semaphore was successfully decremented). This function does **NOT** set *errno*. If an error occurs, the function returns one of the values listed in the Errors section.

Errors:**EAGAIN**

Call would have blocked and *try* was nonzero.

EDEADLK

A deadlock condition was detected.

EINTR

A signal interrupted this function.

EINVAL

The *sync* argument doesn't refer to a valid semaphore.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SyncTypeCreate(), SyncTypeCreate_r()

Create a synchronization object



Don't use the `SyncTypeCreate()` or `SyncTypeCreate_r()` kernel call directly; instead, use the POSIX functions for synchronization objects (see [pthread_cond_init\(\)](#) (p. 2261), [pthread_mutex_init\(\)](#) (p. 2316), [pthread_rwlock_init\(\)](#) (p. 2366), and [sem_init\(\)](#) (p. 2811)).

Synopsis:

```
#include <sys/neutrino.h>

int SyncTypeCreate(
    unsigned type,
    sync_t * sync,
    const struct _sync_attr * attr );

int SyncTypeCreate_r(
    unsigned type,
    sync_t * sync,
    const struct _sync_attr * attr );
```

Arguments:

type

One of the following:

- `_NTO_SYNC_MUTEX_FREE` — create a mutex.
- `_NTO_SYNC_SEM` — create a semaphore.
- `_NTO_SYNC_COND` — create a condition variable.

sync

A pointer to a `sync_t` object that the kernel sets up for the synchronization object; you should treat this as an opaque structure.

attr

A pointer to a `_sync_attr` structure that specifies attributes for the object. This structure contains at least the following members:

- `int __protocol` — `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`.



If you use `PTHREAD_PRIO_PROTECT` and you want to change the priority ceiling to a value above the maximum permitted for unprivileged processes, your process must have the `PROCmgr_AID_PRIORITY` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

If `attr` is `NULL`, the default attributes (`PTHREAD_PRIO_INHERIT`) are assumed.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *`SyncTypeCreate()`* and *`SyncTypeCreate_r()`* kernel calls create a synchronization object in the kernel and initializes *`sync`* for use in other synchronization kernel calls. These functions are similar, except for the way they indicate errors. See the [Returns](#) (p. 3306) section for details.



You should allocate synchronization objects only in normal memory mappings. On certain processors, atomic operations such as calls to [pthread_mutex_lock\(\)](#) (p. 2318) will cause a fault if the control structure is allocated in uncached memory.

Synchronization objects can be used for mutexes, semaphores, or condition variables. You can destroy the synchronization object by calling [SyncDestroy\(\)](#) (p. 3289).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

`SyncTypeCreate()`

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

`SyncTypeCreate_r()`

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function can return any value in the Errors section.

Errors:

EAGAIN

All kernel synchronization objects are in use.

EBUSY

The synchronization object indicated by *sync* was previously initialized but hasn't been destroyed.

EFAULT

A fault occurred when the kernel tried to access *sync* or *attr*.

EINVAL

Either:

- the *type* isn't one of `_NTO_SYNC_COND`, `_NTO_SYNC_MUTEX_FREE` or `_NTO_SYNC_SEM`

Or:

- if the type is correct, and the synchronization object is:
 - a mutex — the protocol isn't one of `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`.
 - a mutex and `PTHREAD_PRIO_PROTECT` is specified — the ceiling priority isn't within the kernel priority range.
 - a condvar — the clock type is invalid.
 - a semaphore — the semaphore value exceeds `SEM_VALUE_MAX`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sysconf()

Return the value of a configurable system limit

Synopsis:

```
#include <unistd.h>
#include <limits.h>

long sysconf( int name );
```

Arguments:

name

The name of the limit that you want to get; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *sysconf()* function returns the value of a configurable system limit indicated by *name*.

Configurable limits are defined in `<confname.h>`, and include at least the following values:

`_SC_AIO_PRIO_DELTA_MAX`

The maximum amount by which a process can decrease its asynchronous I/O priority level from its own scheduling priority.

`_SC_ARG_MAX`

Maximum length of arguments for the *exec*()* functions, in bytes, including environment data.

`_SC_CHILD_MAX`

Maximum number of simultaneous processes per real user ID.

`_SC_CLK_TCK`

The number of intervals per second used to express the value in type `clock_t`.

`_SC_DELAYTIMER_MAX`

The maximum number of times a timer can overrun and you can still detect it.

`_SC_GETGR_R_SIZE_MAX`

If defined (not -1), the maximum size of buffer that you need to supply to [`getgrgid\_r\(\)`](#) (p. 855) for any memory that it needs to allocate.

`_SC_GETPW_R_SIZE_MAX`

If defined (not -1), the maximum size of buffer that you need to supply to [`getpwent\_r\(\)`](#) (p. 932), [`getpwuid\_r\(\)`](#) (p. 940), [`getspent\_r\(\)`](#) (p. 971), or [`getspnam\_r\(\)`](#) (p. 974) for any memory that they need to allocate.

`_SC_JOB_CONTROL`

If this variable is defined, then job control is supported.

`_SC_NGROUPS_MAX`

The maximum number of simultaneous supplementary group IDs per process.

`_SC_OPEN_MAX`

Maximum number of files that one process can have open at any given time.

`_SC_PAGESIZE`

The default size of a thread's guard area.

`_SC_RCT_MEM`

The minimum amount of memory that the system retains as a resource constraint threshold.

`_SC_RCT_SCOID`

The minimum number of connection IDs that a server retains as a resource constraint threshold.

`_SC_SAVED_IDS`

If this variable is defined, then each process has a saved set-user ID and a saved set-group ID.

`_SC_SEM_NSEMS_MAX`

The maximum number of semaphores that one process can have open at a time. The `sysconf()` function returns -1 to indicate that this value is

indeterminate because that limit applies to both named and unnamed semaphores. The kernel allows an arbitrary number of unnamed semaphores (they're kernel synchronization objects, so the number of them is limited only by the amount of available kernel memory).

`_SC_SIGQUEUE_MAX`

The maximum number of outstanding realtime signals sent per process.

`_SC_THREAD_STACK_MIN`

The minimum stack size for a thread.

`_SC_TZNAME_MAX`

The maximum length of the names for time zones. The value is the same as `TZNAME_MAX`.

`_SC_VERSION`

The current POSIX version that is currently supported. A value of 198808L indicates the August (08) 1988 standard, as approved by the IEEE Standards Board.

Returns:

The requested configurable system limit. If *name* isn't defined for the system, -1 is returned.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>

int main( void )
{
    printf( "_SC_ARG_MAX = %ld\n",
           sysconf( _SC_ARG_MAX ) );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sysctl()

Get or set information about the socket manager

Synopsis:

```

#include <sys/param.h>
#include <sys/sysctl.h>

int sysctl( int * name,
            u_int namelen,
            void * oldp,
            size_t * oldlenp,
            void * newp,
            size_t newlen );

```

Arguments:***name***

An array of integers that specifies the *Management Information Base* (MIB) style name of the item that you want to set or get; see below. The minimum number of entries in this array is two; the maximum is CTL_MAXNAME.

namelen

The length of the name.

oldp

NULL, or a pointer to a buffer where the function can store the old value.

oldlenp

NULL, or a pointer to a location that initially specifies the size of the *oldp* buffer. The function changes the value in this location to be the size of the old information stored in the *oldp* buffer.

newp

NULL, or a pointer to a buffer that holds the new value.

newlen

The size of the new value.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `sysctl()` function retrieves information about the socket manager and allows processes with appropriate privileges to set this information. The data available from `sysctl()` consists of integers and tables. You can also get or set data using the `sysctl` utility at the command line.

The state is described using a *Management Information Base* (MIB) style name, specified in *name*, which is a *namelen* length array of integers.



For descriptions of the most commonly used variables, see the entry for `sysctl` in the *Utilities Reference*.

The `sysctl()` function copies the information for the specified variable into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call delivers as much data as fits in the buffer provided and returns with the error code `ENOMEM`. If you don't need the old value, you can set *oldp* and *oldlenp* to `NULL`.

You can determine the size of the available data by calling `sysctl()` with a `NULL` parameter for *oldp*. The function stores the size of the available data in the location pointed to by *oldlenp*. For some operations, the amount of space may change often. For these operations, the system attempts to round up, so that the returned size is large enough for a call to return the data shortly thereafter.

To specify a new value, set *newp* to point to a buffer of length *newlen* from which the requested value is to be taken. If you're not setting a new value, set *newp* to `NULL` and *newlen* to 0.

The top-level names are defined with a `CTL_` prefix in `<sys/sysctl.h>`. The next and subsequent levels down are found in the following header files:

This header file:	Contains definitions for:
<code><sys/sysctl.h></code>	Top-level identifiers
<code><sys/socket.h></code>	Second-level network identifiers
<code><netinet/in.h></code>	Third-level Internet identifiers and fourth-level IP identifiers
<code><netinet/icmp_var.h></code>	Fourth-level ICMP identifiers
<code><netinet/tcp_var.h></code>	Fourth-level TCP identifiers
<code><netinet/udp_var.h></code>	Fourth-level UDP identifiers

The `<sys/sysctl.h>` file defines the top-level identifiers like this:

```
#define CTL_UNSPEC 0 /* unused */
#define CTL_KERN 1 /* "high kernel": proc, limits */
#define CTL_VM 2 /* virtual memory */
#define CTL_VFS 3 /* file system, mount type is next */
#define CTL_NET 4 /* network, see socket.h */
#define CTL_DEBUG 5 /* debugging parameters */
#define CTL_HW 6 /* generic cpu/io */
#define CTL_MACHDEP 7 /* machine dependent */
#define CTL_USER 8 /* user-level */
#define CTL_DDB 9 /* in-kernel debugger */
#define CTL_PROC 10 /* per-proc attr */
#define CTL_VENDOR 11 /* vendor-specific data */
#define CTL_EMUL 12 /* emulation-specific data */
#define CTL_MAXID 13 /* number of valid top-level ids */

#define CTL_NAMES { \
    { 0, 0 }, \
    { "kern", CTLTYPE_NODE }, \
    { "vm", CTLTYPE_NODE }, \
    { "vfs", CTLTYPE_NODE }, \
    { "net", CTLTYPE_NODE }, \
    { "debug", CTLTYPE_NODE }, \
    { "hw", CTLTYPE_NODE }, \
    { "machdep", CTLTYPE_NODE }, \
    { "user", CTLTYPE_NODE }, \
    { "ddb", CTLTYPE_NODE }, \
    { "proc", CTLTYPE_NODE }, \
    { "vendor", CTLTYPE_NODE }, \
    { "emul", CTLTYPE_NODE }, \
}
```

Note how the `CTL_*` definitions correspond to the strings defined below. Let's look at how you'd build the array of integers from a few strings:

kern.clockrate

The top-level name (`kern`) corresponds to `CTL_KERN`, as shown above. The second-level name (`clockrate`) is also defined in `<sys/sysctl.h>`. The entries in the `name` array that you'd pass to `sysctl()` would be as follows:

Entry	Value	See:
0	CTL_KERN	<sys/sysctl.h>
1	KERN_CLOCKRATE	<sys/sysctl.h>

kern.mbuf.mblowat

Once again, the top-level name corresponds to `CTL_KERN`. The second-level name is defined in `<sys/sysctl.h>`, but you need to look in `<sys/mbuf.h>` for the third-level name. The entries in the `name` array would be as follows:

Entry	Value	See:
0	CTL_KERN	<sys/sysctl.h>
1	KERN_MBUF	<sys/sysctl.h>
2	MBUF_MBLowAT	<sys/mbuf.h>

net.inet.tcp.mssdflt

The first-level name (*net*) corresponds to CTL_NET, as shown above. The second-level name (*inet*) is for information about Internet protocols, and is defined as one of the address families in <sys/socket>. The third-level name (*tcp*) is defined as one of the protocols in <netinet/in.h>, and the fourth-level name (*mssdflt*) is defined in <netinet/in_offload.h>.

The entries in the *name* array would be as follows:

Entry	Value	See:
0	CTL_NET	<sys/sysctl.h>
1	AF_INET	<sys/socket.h>
2	IPPROTO_TCP	<netinet/in.h>
3	TCPCTL_MSSDFLT	<netinet/in_offload.h>

Returns:

0

Success.

-1

An error occurred (*errno* (p. 539) is set).

Errors:

EFAULT

One of the buffers *name*, *oldp*, *newp*, or the length pointer *oldlenp* contains an invalid address.

EINVAL

The name array is less than two or greater than CTL_MAXNAME entries long, or a non-NULL *newp* was given, and its specified length in *newlen* is too large or too small.

ENOMEM

The length pointed to by *oldlenp* is too short to hold the requested value.

ENOTDIR

The name array specifies an intermediate rather than terminal name.

EOPNOTSUPP

The name array specifies an unknown value.

EPERM

An attempt was made to set a read-only value, or a process without the appropriate privilege attempted to set or change a value protected by the current system security level.

Examples:

The following code fragment checks whether the UDP packets checksum is enabled:

```
int mib[5], val;
size_t len;

mib[0] = CTL_NET;
mib[1] = AF_INET;
mib[2] = IPPROTO_UDP;
mib[3] = UDPCTL_CHECKSUM;
len = sizeof(val);
sysctl(mib, 4, &val, &len, NULL, 0);
```

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

syslog()

Write a message to the system log

Synopsis:

```
#include <syslog.h>

void syslog( int priority,
             const char * message,
             ...)
```

Arguments:

priority

The priority of the message; see “[Message levels](#) (p. 3316),” below.

message

The message that you want to write. This message is identical to a [printf\(\)](#) (p. 2133)-format string, except that %m is replaced by the current error message (as denoted by the global variable [errno](#) (p. 539)). A trailing newline is added if none is present.

The formatting characters that you use in the message determine any additional arguments.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *syslog()* function writes *message* to the system message logger. The message is then written to the system console, log files, and logged-in users, or forwarded to other machines as appropriate. (See the `syslogd` command.)

The [vsyslog\(\)](#) (p. 3643) function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of `<stdarg.h>`.

Message levels

The message is tagged with *priority*. Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from the following list (ordered from high to low):

LOG_EMERG

A panic condition. This is normally broadcast to all users.

LOG_ALERT

A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT

Critical conditions (for example, hard device errors).

LOG_ERR

General errors.

LOG_WARNING

Warning messages.

LOG_NOTICE

Conditions that aren't error conditions, but should possibly be specially handled.

LOG_INFO

Informational messages.

LOG_DEBUG

Messages that contain information normally of use only when debugging a program.

Examples:

```
syslog(LOG_ALERT, "who: internal error 23");
openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);
syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No

Safety:	
Thread	No

sysmgr_reboot()

Reboot the system

Synopsis:

```
#include <sys/sysmgr.h>

int sysmgr_reboot( void );
```

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *sysmgr_reboot()* function reboots the calling computer. In order to successfully call this function, your process must have the PROCMGR_AID_REBOOT ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

The *sysmgr_reboot()* function doesn't return if successful. If an error occurs, it returns:

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sysmgr_runstate(), sysmgr_runstate_dynamic()

Change the runstate of a processor

Synopsis:

```
#include <sys/sysmgr.h>

int sysmgr_runstate( unsigned cpu,
                    int enable );

int sysmgr_runstate_dynamic( unsigned cpu,
                             int enable );
```

Arguments:

cpu

The index of the CPU whose state you want to change.

enable

0 to turn the processor off; non-zero to turn it on. For more details, see below.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *sysmgr_runstate()* and *sysmgr_runstate_dynamic()* functions turn the specified CPU on or off as follows:

- *sysmgr_runstate()* takes a CPU down manually (if *enable* is zero) or lets it run (if one).
- *sysmgr_runstate_dynamic()* lets the kernel dynamically bring a CPU up or down (if *enable* is one), depending on the system load.

These settings work together:

Runstate	Dynamic runstate	Action
0	—	The CPU is permanently off

Runstate	Dynamic runstate	Action
1	0	The CPU is permanently on (the default)
1	1	The CPU is dynamically brought up or down by the kernel, depending on load.



In order to successfully change the state, your process must have the PROCmgr_AID_RUNSTATE ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Errors:

EINVAL

The CPU index is invalid.

EOK

Success.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

sysmgr_runstate_burst()

Ask the system to turn on dynamically offlined CPUs

Synopsis:

```
#include <sys/sysmgr.h>

int sysmgr_runstate_burst( const unsigned ms_length );
```

Arguments:

ms_length

The length of time, in milliseconds, that the user expects the system to be busy, or 0 to cancel the burst-mode request. The time can be up to 65535 ms.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

Most of the time, the dynamic CPU-onlining algorithm works fine, but it can take up to 64 ms to decide that there's enough system load to turn on a dynamically offlined CPU, and sometimes we just don't want to wait that long. The *sysmgr_runstate_burst()* function tells the kernel that the system is about to get busy, and it should be turning on CPUs much sooner than it would normally.

Multiple threads can call the function, and burst mode remains in effect until the last time period expires. A thread can cancel its burst-mode request by calling *sysmgr_runstate_burst()* again with an argument of 0; any other thread requests remain in force.

In order to successfully call this function, your process must have the `PROCmgr_AID_RUNSTATE_BURST` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

An [errno](#) (p. 539) value if an error occurred.

Classification:*QNX Neutrino* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

SYSPAGE_CPU_ENTRY()

Return a CPU-specific entry from the system page

Synopsis:

```
#include <sys/syspage.h>
#define SYSPAGE_CPU_ENTRY( cpu, entry )...
```

Arguments:

cpu

The CPU to get the entry for.

entry

The entry to get; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SYSPAGE_CPU_ENTRY()` macro returns a pointer to the specified *entry* from the part of the system page that's specific to the given *cpu*.

The best way to reference the system page is via the kernel calls and POSIX cover functions. If there isn't a function to access the information you need, use `SYSPAGE_CPU_ENTRY()` instead of referencing the `_syspage_ptr` variable directly. For information in the rest of the `syspage_entry` structure, use [SYSPAGE_ENTRY\(\)](#) (p. 3326).

Returns:

A pointer to the structure for the given *entry*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

SYSPAGE_CPU_ENTRY() is a macro.

SYSPAGE_ENTRY()

Return an entry from the system page

Synopsis:

```
#include <sys/syspage.h>

#define SYSPAGE_ENTRY( entry )...
```

Arguments:

entry

The entry to get; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SYSPAGE_ENTRY()` macro returns a pointer to the specified *entry* in the system page. For more information, see “Structure of the system page” in the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

The best way to reference the system page is via the kernel calls and POSIX cover functions. If there isn't a function to access the information you need, use `SYSPAGE_ENTRY()` instead of referencing the `_syspage_ptr` variable directly. For information in the CPU-specific part of the `syspage_entry` structure, use [SYSPAGE_CPU_ENTRY\(\)](#) (p. 3324).

The entries you're likely to access with `SYSPAGE_ENTRY()` include:

cpuinfo

Information that's specific to the CPU. For example, to determine if the processor supports NEON technology, a 128-bit SIMD (Single Instruction, Multiple Data) architecture extension to the ARM Cortex-A series processor, do the following:

```
if (SYSPAGE_ENTRY(cpuinfo)->flags & ARM_CPU_FLAG_NEON)
```

qtime

QNX-specific time information. The `qtime_entry` structure contains at least the following members:

- `unsigned long boot_time` — the time, in seconds, since the Unix Epoch (00:00:00 January 1, 1970 Coordinated Universal Time (UTC)) when this system was booted.

If you call [ClockTime\(\)](#) (p. 333) to set the time of day, the kernel checks to see if this field is zero. If it is, the kernel sets it to the appropriate value. There's a -T option for all startup programs that prevents the setting of this field, so that the kernel will set it the first time you call [ClockTime\(\)](#) to change the time of day. This is useful if the RTC hardware isn't in UTC.

- `uint64_t cycles_per_sec` — the number of CPU clock cycles per second for this system. For more information, see [ClockCycles\(\)](#) (p. 325).

Returns:

A pointer to the structure for the given *entry*.

Examples:

```
#include <sys/neutrino.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syspage.h>

int main( void )
{
    uint64_t cps, cycle1, cycle2, ncycles;
    double sec;

    /* snap the time */
    cycle1=ClockCycles( );

    /* do something */
    printf("Hello!\n");

    /* snap the time again */
    cycle2=ClockCycles( );
    ncycles=cycle2-cycle1;
    printf("%lld cycles elapsed.\n", ncycles);

    /* find out how many cycles per second */
    cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    printf( "This system has %lld cycles/sec.\n",cps );
    sec=(double)ncycles/cps;
    printf("The cycles in seconds is %f.\n",sec);

    return EXIT_SUCCESS;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

SYPAGE_ENTRY() is a macro.

_syspage_ptr

A pointer to the system page

Synopsis:

```
#include <sys/syspage.h>

struct syspage_entry *_syspage_ptr
```

Description:

This global variable holds a pointer to the *system page*, which contains information about the system, including the processor type, bus type, and the location and size of available system RAM. For information about this structure, see the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

The best way to reference the system information page is via the kernel calls and POSIX cover functions. If there isn't a function to access the information you need, you should use the [SYSPAGE_ENTRY\(\)](#) (p. 3326) and [SYSPAGE_CPU_ENTRY\(\)](#) (p. 3324) macros instead of referencing the *_syspage_ptr* variable directly.

Classification:

[QNX Neutrino](#) (p. 46)

system()

Execute a system command

Synopsis:

```
#include <stdlib.h>

int system( const char *command );
```

Arguments:

command

NULL, or the system command that you want to execute; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The behavior of the `system()` function depends on the value of its *command* argument:

- If *command* is NULL, `system()` determines whether or not a shell is present.
- If *command* isn't NULL, `system()` invokes a copy of the shell, and passes the string *command* to it for processing. This function uses [spawnlp\(\)](#) (p. 3111) to load a copy of the shell.

The shell used is always `/bin/sh`, regardless of the setting of the ***SHELL*** environment variable, because applications may rely on features of the standard shell, and may fail as a result of running a different shell.



This means that any command that can be entered to the OS can be executed, including programs, QNX Neutrino commands, and shell scripts. The `exec*()` and `spawn*()` functions can only cause programs to be executed.

Returns:

- If *command* is NULL, `system()` returns zero if the shell isn't present, or a nonzero value if the shell is present.

- If *command* isn't NULL, *system()* returns the result of invoking a copy of the shell. If the shell couldn't be loaded, *system()* returns -1; otherwise, it returns the status of the specified command. Use the *WEXITSTATUS()* macro to determine the low-order 8 bits of the termination status of the process.

For example, assume that *status* is the value returned by *system()*. If *WEXITSTATUS(status)* returns 255, either the specified command returned a termination status of 255, or the shell didn't exit (i.e. it died from a signal or couldn't be started at all) and the return value was 255 due to implementation details. For example, under QNX Neutrino and most Unix systems, the value is 255 if *status* is -1, which indicates that the shell couldn't be executed. *WEXITSTATUS()* is defined in `<sys/wait.h>`.

For information about macros that extract information from the value returned by *system()*, see “[Status macros](#) (p. 3652)” in the description of [wait\(\)](#) (p. 3652).

When an error has occurred, [errno](#) (p. 539) contains a value that indicates the type of error that has been detected.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int main( void )
{
    int rc;

    rc = system( "ls" );
    if( rc == -1 ) {
        printf( "shell could not be run\n" );
    } else {
        printf( "result of running command is %d\n",
            WEXITSTATUS( rc ) );
    }
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 22

T

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
thread_pool_attr_t (p. 3388)	thread_pool_create() (p. 3387)
_TRACE_GET_BUFFNUM()	InterruptHookTrace() (p. 1171)
_TRACE_GET_BUFFSEQ()	InterruptHookTrace() (p. 1171)

This chapter describes the API components whose names begin with “T”.

tan(), *tanf()*, *tanl()*

Calculate the tangent of an angle

Synopsis:

```
#include <math.h>

double tan( double x );
float tanf( float x );
long double tanl( long double x );
```

Arguments:

x

The angle, in radians, for which you want to compute the tangent.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the tangent (specified in radians) of *x*. A large magnitude argument may yield a result with little or no significance.

Returns:

The tangent value. If *x* is NAN or infinity, NAN is returned. For a correct value that would cause an underflow, these functions return 0.0.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", tan(.5) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
0.546302
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

tanh(), *tanhf()*, *tanhf()*

Calculate the hyperbolic tangent

Synopsis:

```
#include <math.h>

double tanh( double x );
float tanhf( float x );
long double tanhl( long double x );
```

Arguments:

x

The angle, in radians, for which you want to compute the hyperbolic tangent.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the hyperbolic tangent (specified in radians) of *x*.

When the *x* argument is large, partial or total loss of significance may occur.

Returns:

The hyperbolic tangent value.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", tanh(.5) );
}
```

```
    return EXIT_SUCCESS;  
}
```

produces the output:

```
0.462117
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

tcdrain()

Wait until all output has been transmitted to a device

Synopsis:

```
#include <termios.h>

int tcdrain( int fildes );
```

Arguments:

fildes

A file descriptor that's associated with the device that you want to wait for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcdrain()* function waits until all output has been physically transmitted to the device associated with *fildes*, or until a signal is received.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The argument *fildes* is invalid.

EINTR

A signal interrupted the operation.

ENOSYS

The resource manager associated with *fildes* doesn't support this call.

ENOTTY

The argument *fildes* doesn't refer to a terminal device.

Examples:

```
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fildes;

    fildes = open( "/dev/ser1", O_RDWR );
    write( fildes, "ATH", 3 );

    /* Wait for data to transmit before returning */
    tcdrain( fildes );
    close( fildes );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcdropline()

Disconnect a communications line

Synopsis:

```
#include <termios.h>

int tcdropline( int fd,
                int duration );
```

Arguments:

fd

A file descriptor that's associated with the line that you want to disconnect.

duration

The number of milliseconds that you want to drop the line for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcdropline()* function initiates a disconnect condition on the communication line associated with the opened file descriptor indicated by *fd*.

The disconnect condition lasts at least *duration* milliseconds, or approximately 300 milliseconds if *duration* is zero. The system rounds the effective value of *duration* up to the next highest supported interval, which is typically a multiple of 100 milliseconds.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid *fd* argument.

ENOSYS

The resource manager associated with *fd* doesn't support this call.

ENOTTY

The argument *fd* doesn't refer to a terminal device.

Examples:

```
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    fd = open( "/dev/ser1", O_RDWR );

    /* Disconnect for 500 milliseconds */
    tcdropline( fd, 500 );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcflow()

Perform a flow-control operation on a data stream

Synopsis:

```
#include <termios.h>

int tcflow( int fildes,
            int action );
```

Arguments:

fildes

A file descriptor that's associated with the data stream that you want to perform the operation on.

action

The action you want to perform; see below.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tcflow()* function performs a flow-control operation on the data stream associated with *fildes*, depending on the values in *action*.

At least the following actions are defined in `<termios.h>`:

TCOOFF

Use software flow control to suspend output on the device associated with *fildes*.

TCOOFFHW

Use hardware flow control to suspend output on the device associated with *fildes*.

TCOON

Use software flow control to resume output on the device associated with *fildev*.

TCOONHW

Use hardware flow control to resume output on the device associated with *fildev*.

TCIOFF

Cause input to be flow-controlled by sending a STOP character immediately across the communication line associated with *fildev*, (that is, software flow control).

TCIOFFHW

Cause input to be flow-controlled by using hardware control.

TCION

Resume input by sending a START character immediately across the communication line associated with *fildev* (that is, software flow control).

TCIONHW

Cause input to be resumed by using hardware flow control.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

Invalid *fildev* argument.

EINVAL

Invalid *action* argument.

ENOSYS

The resource manager associated with *fildev* doesn't support this call.

ENOTTY

The argument *fildev* doesn't refer to a terminal device.

Examples:

```
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    fd = open( "/dev/ser1", O_RDWR );

    /* Resume output on flow-controlled device */
    tcflow( fd, TCOON );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcflush()

Flush the input and/or output stream

Synopsis:

```
#include <termios.h>

int tcflush( int fildes,
             int queue_selector );
```

Arguments:

fildes

A file descriptor that's associated with the data stream that you want to perform the operation on.

queue_selector

The stream or streams that you want to flush. At least the following values for *queue_selector* are defined in <termios.h>:

- **TCIFLUSH** — discard all data that's received, but not yet read, on the device associated with *fildes*.
- **TCOFLUSH** — discard all data that's written, but not yet transmitted, on the device associated with *fildes*.
- **TCIOFLUSH** — discard all data that's written, but not yet transmitted, as well as all data that's received, but not yet read, on the device associated with *fildes*.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcflush()* function flushes the input stream, the output stream, or both, depending on the value of the argument *queue_selector*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

Invalid *fil*des argument.

EINVAL

Invalid *queue_selector* argument.

ENOSYS

The resource manager associated with *fil*des doesn't support this call.

ENOTTY

The argument *fil*des doesn't refer to a terminal device.

Examples:

```
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fildes;

    fildes = open( "/dev/ser1", O_RDWR );

    /* Throw away all input data */
    tcflush( fildes, TCIFLUSH );

    close( fildes );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcgetattr()

Get the current terminal control settings

Synopsis:

```
#include <termios.h>

int tcgetattr( int fildes,
               struct termios *termios_p );
```

Arguments:

fildes

The file descriptor associated with the terminal device.

termios_p

A pointer to a [termios](#) (p. 3379) structure in which *tcgetattr()* can store the terminal's control attributes.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tcgetattr()* function gets the current terminal control settings for the opened device indicated by *fildes*, and stores the results in the structure pointed to by *termios_p*.

For more information, see chapter 7 of POSIX 1003.1

Returns:

0

Success.

-1

An error occurred; *errno* is set.

Errors:

EBADF

The *fildev* argument is invalid.

ENOSYS

The resource manager associated with *fildev* doesn't support this call.

ENOTTY

The *fildev* argument doesn't refer to a terminal device.

Examples:

See [tcsetattr\(\)](#) (p. 3364).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcgetpgrp()

Get the process group ID associated with a device

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp( int fildes );
```

Arguments:***fildes***

A file descriptor that's associated with the device whose process group ID you want to get.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `tcgetpgrp()` function returns the process group ID of the foreground process that's associated with the device indicated by *fildes*.

Returns:

The ID of foreground process group. If an error occurs, -1 is returned, and [errno](#) (p. 539) is set.

Errors:**EBADF**

The argument *fildes* is invalid.

ENOSYS

The resource manager associated with *fildes* doesn't support this call.

ENOTTY

The argument *fildes* isn't associated with a terminal device.

Examples:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "STDIN directs breaks to pgrp %d\n",
           tcgetpgrp( 0 ) );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcgetsid()

Get the process group ID of the session leader for a controlling terminal

Synopsis:

```
#include <unistd.h>

int tcgetsid( int filedes );
```

Arguments:

filedes

A file descriptor that's associated with the device whose ID you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcgetsid()* function returns the process group ID of the session for which the terminal specified by *filedes* is the controlling terminal.

Returns:

The process group ID associated with the terminal, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**EACCES**

The *filedes* argument isn't associated with a controlling terminal.

EBADF

The *filedes* argument isn't a valid file descriptor.

ENOTTY

The file associated with *filedes* isn't a terminal.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcgetsize()

Get the size of a character device

Synopsis:

```
#include <termios.h>

int tcgetsize( int filedes,
               int* prows,
               int* pcols );
```

Arguments:

filedes

A file descriptor that's associated with the device whose size you want to get.

prows, *pcols*

NULL, or pointers to locations where the function can store the number of rows and columns.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcgetsize()* function gets the size of the character device associated with *filedes* and stores the number of rows and columns in *prows* and *pcols* if they're not NULL.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

The *filedes* argument isn't associated with a controlling terminal.

EBADF

The *filedes* argument isn't a valid file descriptor.

ENOTTY

The file associated with *filedes* isn't a terminal.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcinject()

Inject characters into a device's input buffer

Synopsis:

```
#include <termios.h>

int tcinject( int fd,
              char *buf,
              int n );
```

Arguments:

fd

A file descriptor that's associated with the device whose input buffer you want to add characters to.

buf

A pointer to a buffer that contains the characters that you want to insert.

n

The number of characters to insert. If *n* is positive, the characters are written to the canonical (edited) queue. If *n* is negative, the characters are written to the raw queue.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `tcinject()` function injects *n* characters pointed to by *buf* into the input buffer of the device given in *fd*.

Note that while injecting into the canonical queue, editing characters in *buf* are acted upon as though the user entered them directly from the device. If *buf* doesn't contain a newline (`'\n'`), carriage return (`'\r'`) or a data-forwarding character such as an EOF, data doesn't become available for reading. If *buf* does contain a data-forwarding character, it should contain only one as the last character in *buf*.

This function is useful for implementing command-line recall algorithms by injecting recalled lines into the canonical queue.

Returns:

- 0
Success.
- 1
An error occurred ([errno](#) (p. 539) is set).

Errors:

- EBADF**
The *fd* argument is invalid or the file isn't opened for read.
- ENOSYS**
This function isn't supported for the device opened.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

int main( void )
{
    char *p = "echo Hello world!\n";

    /* Inject the line all at once */
    tcinject(0, p, strlen(p));

    /* Inject the line one character at a time */
    while(*p)
        tcinject(0, p++, 1);
    return EXIT_SUCCESS;
}
```

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcischars()

Determine the number of characters waiting to be read

Synopsis:

```
#include <termios.h>

int tcischars( int filedes );
```

Arguments:

filedes

A file descriptor that's associated with the device that you want to check.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcischars()* function checks to see how many characters are waiting to be read from the given file descriptor, *filedes*.

Returns:

The number of characters waiting to be read, or -1 if an error occurred.

Errors:

ENOTTY

The *fd* argument isn't the file descriptor for a character device.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

TCP

Internet Transmission Control Protocol

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_STREAM,
            0 );
```

Description:

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It's a byte-stream protocol used to support the `SOCK_STREAM` abstraction.

TCP uses the standard Internet address format and also provides a per-host collection of “port addresses.” Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets using the TCP protocol are either *active* or *passive*. Active sockets initiate connections to passive sockets. By default, TCP sockets are created active.

To create a passive socket, you must bind the socket with the [bind\(\)](#) (p. 208) system call, and then use the [listen\(\)](#) (p. 1523) system call. Only passive sockets may use the [accept\(\)](#) (p. 58) call to accept incoming connections; only active sockets may use the [connect\(\)](#) (p. 348) call to initiate connections.

Passive sockets may “underspecify” their location to match incoming connection requests from multiple networks. With this technique, termed *wildcard addressing*, a single server can provide service to clients on multiple networks. If you wish to create a socket that listens on all networks, the Internet address `INADDR_ANY` must be bound. You can still specify the TCP port at this time. If the port isn't specified, the system assigns one.

Once a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network.

TCP supports several socket options (defined in `<netinet/tcp.h>`) that you can set with [setsockopt\(\)](#) (p. 2909) and retrieve with [getsockopt\(\)](#) (p. 962). The option level for these calls is the protocol number for TCP, available from [getprotobyname\(\)](#) (p. 924).

TCP_NODELAY

Under most circumstances, TCP sends data when it's presented. When outstanding data hasn't yet been acknowledged, TCP gathers small amounts of output to be sent in a single packet once an acknowledgment is received.

For a few clients (such as windowing systems that send a stream of mouse events that receive no replies), this packetization may cause significant delays. Therefore, TCP provides a boolean option, `TCP_NODELAY`, to defeat this algorithm.

TCP_MAXSEG

The Maximum Segment Size (MSS) for a TCP connection. The value returned is the maximum amount of data that TCP sends to the other end. If this value is fetched before the socket is connected, the value returned is the default value that's used if an MSS option isn't received from the other end.

TCP_KEEPALIVE

Specifies the idle time in seconds for the connection before TCP starts sending “keepalive” probes. The default value is 2 hours. This option is effective only when the `SO_KEEPALIVE` socket option is enabled.

You can use options at the IP transport level with TCP (see the [IP](#) (p. 1391) protocol. Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

Based on:

RFC 793

Returns:

A descriptor referencing the socket, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EADDRINUSE

You tried to create a socket with a port that's already been allocated.

EADDRNOTAVAIL

You tried to create a socket with a network address for which no network interface exists.

ECONNREFUSED

The remote peer actively refused connection establishment (usually because no process was listening to the port).

ECONNRESET

The remote peer forced the connection to be closed.

EISCONN

You tried to establish a connection on a socket that already has one.

ENOBUFS

The system ran out of memory for an internal data structure.

ETIMEDOUT

A connection was dropped due to excessive retransmissions.

tcsendbreak()

Assert a break condition over a communications line

Synopsis:

```
#include <termios.h>

int tcsendbreak( int fildev,
                 int duration );
```

Arguments:

fd

A file descriptor that's associated with the line that you want to assert the break on.

duration

The number of milliseconds that you want to break for.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tcsendbreak()* function asserts a break condition over the communication line associated with the opened file descriptor indicated by *fildev*.

The break condition lasts for at least *duration* milliseconds, or approximately 300 milliseconds if *duration* is zero. The system rounds the effective value of *duration* up to the next highest supported interval, which is typically a multiple of 100 milliseconds.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EBADF**

The argument *fildev* is invalid.

ENOSYS

The resource manager associated with *fildev* doesn't support this call.

ENOTTY

The argument *fildev* doesn't refer to a terminal device.

Examples:

```
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    fd = open( "/dev/ser1", O_RDWR );

    /* Send a 500 millisecond break */
    tcsendbreak( fd, 500 );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcsetattr()

Change the terminal control settings for a device

Synopsis:

```
#include <termios.h>

int tcsetattr( int fildes,
               int optional_actions,
               const struct termios *termios_p );
```

Arguments:

fildes

The file descriptor associated with the terminal device.

termios_p

A pointer to a [termios](#) (p. 3379) structure that describes the attributes that you want to set for the terminal device.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tcsetattr()* function sets the current terminal control settings for the opened device indicated by *fildes* to the values stored in the structure pointed to by *termios_p*.

The operation of *tcsetattr()* depends on the values in *optional_actions*:

TCSANOW

The change is made immediately.

TCSADRAIN

No change is made until all currently written data has been transmitted.

TCSAFLUSH

No change is made until all currently written data has been transmitted, at which point any received but unread data is also discarded.

The `termios` control structure is defined in `<termios.h>`. For more information, see [tcgetattr\(\)](#) (p. 3347).

Returns:

0

Success.

-1

An error occurred; *errno* is set.

Errors:

EBADF

The argument *fildev* is invalid;

EINVAL

The argument *action* is invalid, or one of the members of *termios_p* is invalid.

ENOSYS

The resource manager associated with *fildev* doesn't support this call.

ENOTTY

The argument *fildev* doesn't refer to a terminal device.

Examples:

```
#include <stdlib.h>
#include <termios.h>

int raw( int fd )
{
    struct termios termios_p;

    if( tcgetattr( fd, &termios_p ) )
        return( -1 );

    termios_p.c_cc[VMIN] = 1;
    termios_p.c_cc[VTIME] = 0;
    termios_p.c_lflag &= ~( ECHO|ICANON|ISIG|
        ECHOE|ECHOK|ECHONL );
    termios_p.c_oflag &= ~( OPOST );
    return( tcsetattr( fd, TCSADRAIN, &termios_p ) );
}

int unraw( int fd )
{
    struct termios termios_p;

    if( tcgetattr( fd, &termios_p ) )
        return( -1 );

    termios_p.c_lflag |= ( ECHO|ICANON|ISIG|
        ECHOE|ECHOK|ECHONL );
    termios_p.c_oflag |= ( OPOST );
    return( tcsetattr( fd, TCSADRAIN, &termios_p ) );
}
```

```
int main( void )
{
    raw( 0 );
    /*
     * Stdin is now "raw"
     */
    unraw ( 0 );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcsetpgrp()

Set the process group ID for a device

Synopsis:

```
#include <sys/types.h>
#include <unistd.h>

int tcsetpgrp( int fildes,
               pid_t pgrp_id );
```

Arguments:

fildes

A file descriptor that's associated with the device whose process group ID you want to set.

pgrp_id

The process group ID that you want to assign to the device.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tcsetpgrp()* function sets the process group ID associated with the device indicated by *fildes* to be *pgrp_id*.



In order to set the process group ID, your process must have the `PROC_MGR_AID_SESSION` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

If successful, the *tcsetpgrp()* function causes subsequent breaks on the indicated terminal device to generate a `SIGINT` on all process in the given process group.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EBADF

The argument *fildev* is invalid.

EINVAL

The argument *pgroup_id* is invalid.

ENOSYS

The resource manager associated with *fildev* doesn't support this call.

ENOTTY

The argument *fildev* isn't associated with a terminal device.

EPERM

The argument *pgroup_id* isn't part of the same session as the calling process.

Examples:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    /*
     * Direct breaks on stdin to me
     */
    tcsetpgrp( 0, getpid() );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tcsetsid()

Make a terminal device a controlling device

Synopsis:

```
#include <termios.h>

int tcsetsid( const int fd,
              const pid_t pid );
```

Arguments:

fd

A file descriptor that's associated with the device that you want to make a controlling device.

pid

The ID of the process that you want to associate with the controlling device.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `tcsetsid()` function makes the terminal device associated with the file descriptor argument *fd* into a controlling terminal that's associated with the process *pid*. If successful, this call causes subsequent hangup conditions on the terminal device *fd* to generate a **SIGHUP** signal on the given process.

This call is equivalent to calling `ioctl( fd, TIOCSCTTY )` to set the controlling terminal to the current process. You can clear the controlling terminal by passing `-1` as *fd*.

Returns:

0

Success.

-1

Failure; [errno](#) (p. 539) is set.

Errors:**EBADF**

Invalid file descriptor.

EINVAL

The argument *pid* is invalid.

ENOSYS, ENOTTY

The argument *fd* isn't associated with a terminal device.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

tcsetattr()

Set the size of a character device

Synopsis:

```
#include <termios.h>

int tcsetattr( int filedes,
               int rows,
               int cols );
```

Arguments:

filedes

A file descriptor that's associated with the device whose size you want to set.

rows, cols

The number of rows and columns that you want to use.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `tcsetattr()` function sets the size of the character device associated with *filedes* to the given number of rows and columns.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EACCES

The *filedes* argument isn't associated with a controlling terminal.

EBADF

The *filedes* argument isn't a valid file descriptor.

EINVAL

The *rows* or *cols* argument is invalid.

ENOTTY

The file associated with *filedes* isn't a terminal.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tell(), tell64()

Determine the current file position

Synopsis:

```
#include <unistd.h>

off_t tell( int filedes );
off64_t tell64( int filedes );
```

Arguments:

filedes

The file descriptor of the file whose position you want to get.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tell()* and *tell64()* functions determine the current file position for any subsequent *read()* or *write()* operation (that is, any subsequent unbuffered file operation). The *tell64()* function is a 64-bit version of *tell64()*. The *filedes* value is the file descriptor returned by a successful call to *open()* (p. 1905).

You can use the returned value in conjunction with *lseek()* (p. 1555) to reset the current file position.

Returns:

The current file position, expressed as the number of bytes from the start of the file, or -1 if an error occurs (*errno* (p. 539) is set). A value of 0 indicates the start of the file.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

char buffer[]
```

```
        = { "A text record to be written" };

int main( void )
{
    int filedes;
    int size_written;

    /* open a file for output */
    /* replace existing file if it exists */
    filedes = open( "file",
        O_WRONLY | O_CREAT | O_TRUNC,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    if( filedes != -1 ) {

        /* print file position */
        printf( "%ld\n", tell( filedes ) );

        /* write the text */
        size_written = write( filedes, buffer,
            sizeof( buffer ) );

        /* print file position */
        printf( "%ld\n", tell( filedes ) );

        /* close the file */
        close( filedes );
    }
    return EXIT_SUCCESS;
}
```

produces the output:

```
0
28
```

Classification:

tell() is [QNX 4](#) (p. 46); *tell64()* is [QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

telldir()

Get the location associated with the directory stream

Synopsis:

```
#include <dirent.h>

long int telldir( DIR * dirp );
```

Arguments:

dirp

The directory stream for which you want to get the current location.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *telldir()* function obtains the current location associated with the directory stream specified by *dirp*.

Values returned by *telldir()* are good only for the lifetime of the `DIR` pointer, *dirp*, from which they're derived. If you call [rewinddir\(\)](#) (p. 2648) on the directory stream, or if you close and reopen the directory, an offset previously returned by *telldir()* becomes invalid.

Returns:

The current position of the specified directory stream, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EBADF

The *dirp* argument doesn't refer to an open directory stream.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

tempnam()

Create a name for a temporary file

Synopsis:

```
#include <stdio.h>

char* tempnam( const char* dir,
               const char* px );
```

Arguments:

dir

NULL, or the directory to use in the pathname.

px

NULL, or a prefix to use in the pathname.



If *px* isn't NULL, the string it points to must be no more than 5 bytes long.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tempnam()* function generates a pathname for use as a temporary file. The pathname is in the directory specified by *dir* and has the prefix specified in *px*.

If *dir* is NULL, the pathname is prefixed with the first accessible directory contained in:

- the temporary file directory *P_tmpdir* (defined in `<stdio.h>`)
- the ***TMPDIR*** environment variable
- the `_PATH_TMP` constant (defined in `<paths.h>`).

If all of these paths are inaccessible, *tempnam()* attempts to use `/tmp` and then the current working directory.

The *tempnam()* function generates up to TMP_MAX unique file names before it starts to recycle them.

Returns:

A pointer to the generated file name, which you should deallocate with the [free\(\)](#) (p. 717) function when the application no longer needs it, or NULL if an error occurs.

Errors:**ENOMEM**

There's insufficient memory available to create the pathname.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

Caveats:

The *tempnam()* functions creates only pathnames; the application must create and remove the files.

It's possible for another thread or process to create a file with the same name between the time the pathname is created and the file is opened.

termios

Terminal control structure

Synopsis:

```
struct termios {
    tcflag_t      c_iflag;
    tcflag_t      c_oflag;
    tcflag_t      c_cflag;
    tcflag_t      c_lflag;
    cc_t          c_cc[NCCS];
    uint32_t      reserved[3];
    speed_t       c_ispeed;
    speed_t       c_ospeed;
};
```

Description:

The `termios` control structure is defined in `<termios.h>`, and contains at least the members described below.

tcflag_t c_iflag

Input modes. This member contains at least the following bits:

BRKINT

Signal interrupt on break.

ICRNL

Map CR to NL on input.

IGNBRK

Ignore break conditions.

IGNCR

Ignore CR.

IGNPAR

Ignore characters with parity errors.

INLCR

Map NL to CR on input.

INPCK

Enable input parity check.

ISTRIP

Strip top bit from character.

IXOFF

Enable software input flow control (via START/STOP chars).

IXON

Enable software output flow control (via START/STOP chars).

PARMRK

Mark parity errors in the input data stream.

tcflag_t c_oflag

Output modes. This member contains at least the following bits:

OPOST

Perform output processing.

tcflag_t c_cflag

Control modes. This member contains at least the following bits:

CLOCAL

Ignore modem status lines.

CREAD

Enable receiver.

CSIZE

Number of data bits per character.

CS5

5 data bits.

CS6

6 data bits.

CS7

7 data bits.

CS8

8 data bits.

CSTOPB

Two stop bits, else one.

HUPCL

Hang up on last close.

IHFLOW

Support input flow control using the hardware handshaking lines.

OHFLOW

Support output flow control using the hardware handshaking lines.

PARENB

Parity enable.

PARODD

Odd parity, else even.

PARSTK

Stick parity (mark parity if PARODD is set, else space parity).

`tcflag_t c_lflag`

Local modes. This member contains at least the following bits:

ECHO

Enable echo.

ECHOE

Echo ERASE as destructive backspace.

ECHOK

Echo KILL as a line erase.

ECHONL

Echo '\n', even if ECHO is off.

ICANON

Canonical input mode (line editing enabled).

IEXTEN

QNX Neutrino extensions to POSIX are enabled.

ISIG

Enable signals.

NOFLSH

Disable flush after interrupt, quit, or suspend.

TOSTOP

Send SIGTTOU for background output.

cc_t c_cc[NCCS]

Control characters. The array *c_cc* includes at least the following control characters:

***c_cc*[VEOF]**

EOF character.

***c_cc*[VEOL]**

EOL character.

***c_cc*[VERASE]**

ERASE character.

***c_cc*[VFWD]**

FORWARD character.

***c_cc*[VINTR]**

INTR character.

***c_cc*[VKILL]**

KILL character.

***c_cc*[VMIN]**

MIN value.

***c_cc*[VQUIT]**

QUIT character.

***c_cc*[VSUSP]**

SUSP character.

***c_cc*[VTIME]**

TIME value.

***c_cc*[VSTART]**

START character.

***c_cc*[VSTOP]**

STOP character.

The following control characters are also defined, but are only acted on if they're immediately preceded by the nonzero characters in *c_cc*[VPREFIX][4], and are immediately followed by the nonzero characters in *c_cc*[VSUFFIX][4] and the *IEXTEN* bit of *c_lflag* is set:

***c_cc*[VLEFT]**

Left cursor motion.

***c_cc*[VRIGHT]**

Right cursor motion.

***c_cc*[VUP]**

Up cursor motion.

***c_cc*[VDOWN]**

Down cursor motion.

***c_cc*[VINS]**

Insert character.

***c_cc*[VDEL]**

Delete character.

***c_cc*[VRUB]**

Rubout character.

***c_cc*[VCAN]**

Cancel character.

***c_cc*[VHOME]**

Home character.

c_cc[VEND]

End character.

Any of the control characters in the `c_cc` array can be disabled by setting that character to the `_PC_VDISABLE` parameter which is returned by [*fpathconf\(\)*](#) (p. 699) (typically a zero).

speed_t c_ispeed

Input baud rate. This member should be queried and set with the [*cfgetispeed\(\)*](#) (p. 262) and [*cfsetispeed\(\)*](#) (p. 274) functions.

speed_t c_ospeed

Output baud rate. This member should be queried and set with the [*cfgetospeed\(\)*](#) (p. 264) and [*cfsetospeed\(\)*](#) (p. 276) functions.

Classification:

[*POSIX 1003.1*](#) (p. 46)

thread_pool_control()

Control the thread pool behavior

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

thread_pool_control( thread_pool_t * pool,
                    thread_pool_attr_t * attr,
                    uint16_t lower,
                    uint16_t upper,
                    unsigned flags )
```

Arguments:

pool

A thread pool handle that was returned by [thread_pool_create\(\)](#) (p. 3387).

attr

A pointer to a `thread_pool_attr_t` structure that specifies the attributes that you want to use for the thread pool. For more information, see “[Thread-pool attributes](#) (p. 3388),” in the documentation for [thread_pool_create\(\)](#).

lower, upper

This function blocks until the number of threads created is between the range of *upper* and *lower*, unless you set `THREAD_POOL_CONTROL_NONBLOCK` in *flags*.

flags

Which attributes you want to change for the thread pool; any combination of the following bits:

- `THREAD_POOL_CONTROL_HIWATER` — adjust the high-water value of the number of threads allowed in the thread pool.
- `THREAD_POOL_CONTROL_INCREMENT` — adjust the increment value of the number of threads.
- `THREAD_POOL_CONTROL_LOWATER` — adjust the low-water value of the number of threads allowed in the thread pool.
- `THREAD_POOL_CONTROL_MAXIMUM` — adjust the maximum value of the number of threads allowed in the thread pool.

- `THREAD_POOL_CONTROL_NONBLOCK` — don't block while creating threads.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

Use *thread_pool_control()* to specify a thread pool's behavior and adjust its attributes.



Having several threads call this function with the same thread pool handle isn't recommended.

Returns:

-1 if an error occurs (*errno* (p. 539) is set).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

thread_pool_create()

Create a thread pool handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

thread_pool_t * thread_pool_create (
    thread_pool_attr_t * pool_attr,
    unsigned flags );
```

Arguments:

pool_attr

A pointer to a `thread_pool_attr_t` structure that specifies the attributes that you want to use for the thread pool. For more information, see “[Thread-pool attributes](#) (p. 3388),” below.

flags

Flags (defined in `<sys/dispatch.h>`) that affect what happens to the thread that's creating the pool:

- `POOL_FLAG_EXIT_SELF` — when the pool is started using [thread_pool_start\(\)](#) (p. 3397), exit the thread that called this function.
- `POOL_FLAG_USE_SELF` — when the pool is started, use the calling thread as part of the pool.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `thread_pool_create()` function creates a thread pool handle. This handle is then used to start a thread pool with [thread_pool_start\(\)](#) (p. 3397). With the thread pool functions, you can create and manage a pool of worker threads.

How it works

The worker threads work in the following way:

- When a new worker thread is created, a context is allocated, which the thread uses to do its work.
- The thread then calls the blocking function. This function blocks until the thread has work to do. For example, the blocking function could call [MsgReceive\(\)](#) (p. 1760) to wait for a message.
- After the blocking function returns, the worker thread calls the handler function, which performs the actual work.
- When the handler function returns, the thread calls the blocking function again.

The thread continues to block and handle events until the thread pool decides this worker thread is no longer needed. Finally, when the worker thread exits, it releases the allocated context.

The thread pool manages these worker threads so that there's a certain number of them in the blocked state. Thus, as threads become busy in the handler function, the thread pool creates new threads to keep a minimum number of threads in a state where they can accept requests from clients. By the same token, if the demand on the thread pool goes down, the thread pool lets some of these blocked threads exit.

Thread-pool attributes

The *pool_attr* argument sets the:

- functions that get called, when a new thread is started or one dies, to allocate and free contexts used by threads
- blocking and handler functions
- parameters of the thread pool such as the number of worker threads, etc.

The `thread_pool_attr_t` structure that the *pool_attr* argument points to is defined in `<sys/dispatch.h>` as:

```
typedef struct _thread_pool_attr {
    THREAD_POOL_HANDLE_T    *handle;
    THREAD_POOL_PARAM_T     *(*block_func)
                           (THREAD_POOL_PARAM_T *ctp);
    void                    (*unblock_func)
                           (THREAD_POOL_PARAM_T *ctp);
    int                     (*handler_func)
                           (THREAD_POOL_PARAM_T *ctp);
    THREAD_POOL_PARAM_T     *(*context_alloc)
                           (THREAD_POOL_HANDLE_T *handle);
    void                    (*context_free)
                           (THREAD_POOL_PARAM_T *ctp);
    pthread_attr_t          *attr;
    unsigned short          lo_water;
    unsigned short          increment;
    unsigned short          hi_water;
    unsigned short          maximum;
    const char              *tid_name;
    unsigned                reserved[7];
} thread_pool_attr_t;
```

The members include:

handle

A handle that gets passed to the *context_alloc* function.

block_func

The function that's called when the worker thread is ready to block, waiting for work. This function returns a pointer that's passed to *handler_func*.

unblock_func

The function that's called to unblock threads. If you use [dispatch_block\(\)](#) (p. 426) as the *block_func*, use [dispatch_unblock\(\)](#) (p. 445) as the *unblock_func*.

handler_func

The function that's called after *block_func* returns to do some work. The function is passed the pointer returned by *block_func*.

context_alloc

The function that's called when a new thread is created by the thread pool. It is passed *handle*. The function returns a pointer, which is then passed to the blocking function, *block_func*.

context_free

The function that's called when the worker thread exits, to free the context allocated with *context_alloc*.

attr

A pointer to a [pthread_attr_*](#) (p. 2213) function that's passed to [pthread_create\(\)](#) (p. 2284). The *pthread_attr_** functions set the stack size, priority, etc. of the worker threads. If NULL, default values are used.

lo_water

The minimum number of threads that the pool should keep in the blocked state (i.e. threads that are ready to do work).

increment

The number of new threads created at one time.

hi_water

The maximum number of threads to keep in a blocked state.

maximum

The maximum number of threads that the pool can create.

tid_name

NULL, or a pointer to a null-terminated name for the threads in the pool. If set, this string is passed to [pthread_setname_np\(\)](#) (p. 2398) when the thread pool creates a new thread.

The scope of the *tid_name* string must match the lifetime of the thread pool itself. For example, this is valid:

```
pool_attr.tid_name = "fsys_resmgr";
```



but using a local or automatic variable like this isn't:

```
{
    char name[32];
    snprintf(name, sizeof(name), "cam %d:%d", cam.path, cam.target);
    pool_attr.tid_name = name;
    ...
    return;
}
```

Returns:

A thread pool handle, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

ENOMEM

Insufficient memory to allocate internal data structures.

Examples:

Here's a simple multithreaded resource manager:

```
/* Define an appropriate interrupt number: */
#define INTNUM 0

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>

static resmgr_connect_funcs_t connect_funcs;
static resmgr_io_funcs_t io_funcs;
static iofunc_attr_t attr;

void *interrupt_thread( void *data)
/* *data isn't used */
{
    struct sigevent event;
    int id;

    /* fill in "event" structure */
    memset( &event, 0, sizeof(event) );
    event.sigev_notify = SIGEV_INTR;

    /* INTNUM is the desired interrupt level */
    id = InterruptAttachEvent( INTNUM, &event, 0 );

    ...
}
```

```

    while (1) {
        InterruptWait( 0, NULL );
        /*
         * do something about the interrupt,
         * perhaps updating some shared
         * structures in the resource manager
         *
         * unmask the interrupt when done
         */
        InterruptUnmask( INTNUM, id );
    }
}

int
main(int argc, char **argv) {
    thread_pool_attr_t    pool_attr;
    thread_pool_t         *tpp;
    dispatch_t            *dpp;
    resmgr_attr_t         resmgr_attr;
    int                   id;

    if((dpp = dispatch_create()) == NULL) {
        fprintf( stderr,
            "%s: Unable to allocate dispatch handle.\n",
            argv[0] );
        return EXIT_FAILURE;
    }

    memset( &pool_attr, 0, sizeof pool_attr );
    pool_attr.handle = dpp;
    pool_attr.context_alloc = (void *) dispatch_context_alloc;
    pool_attr.block_func = (void *) dispatch_block;
    pool_attr.unblock_func = (void *) dispatch_unblock;
    pool_attr.handler_func = (void *) dispatch_handler;
    pool_attr.context_free = (void *) dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;
    pool_attr.tid_name = "my_thread_pool";

    if((tpp = thread_pool_create( &pool_attr,
        POOL_FLAG_EXIT_SELF)) == NULL ) {
        fprintf(stderr,
            "%s: Unable to initialize thread pool.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    iofunc_func_init( _RESMGR_CONNECT_NFUNCS,
        &connect_funcs,
        _RESMGR_IO_NFUNCS, &io_funcs );
    iofunc_attr_init( &attr, S_IFNAM | 0666, 0, 0 );

    memset( &resmgr_attr, 0, sizeof resmgr_attr );
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    if((id = resmgr_attach( dpp, &resmgr_attr,
        "/dev/mynull",
        _FTYPE_ANY, 0, &connect_funcs,
        &io_funcs,
        &attr )) == -1) {
        fprintf( stderr,
            "%s: Unable to attach name.\n", argv[0] );
        return EXIT_FAILURE;
    }

    /* Start the thread which will handle interrupt events. */
    pthread_create ( NULL, NULL, interrupt_thread, NULL );

    /* Never returns */
    thread_pool_start( tpp );
    return EXIT_SUCCESS;
}

```

For more examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), and [resmgr_attach\(\)](#) (p. 2580). For information on advanced topics in designing and implementing a resource manager, see the Combine Messages chapter of *Writing a Resource Manager*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

thread_pool_destroy()

Free the memory allocated to a thread pool

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int thread_pool_destroy( thread_pool_t * pool );
```

Arguments:

pool

A thread pool handle that was returned by [thread_pool_create\(\)](#) (p. 3387).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `thread_pool_destroy()` function frees the memory allocated to a thread pool that's identified by the handle `pool`. This is done only after all the threads of the thread pool have exited.



Prior to QNX Neutrino 6.1.0, this function simply deallocated the thread pool handle and returned. Although this was acceptable for servers that never exited (and consequently never shut down their thread pools), it's unsuitable for closing down a thread pool.

The `thread_pool_destroy()` function calls the unblock handler provided in the pool attribute structure. The unblock handler is called at least once for every thread in the thread pool. Once the unblock handler is called, the thread calling `thread_pool_destroy()` blocks until the number of threads in the thread pool drops to zero. When there are no more threads in the thread pool, the handle `pool` is freed and `thread_pool_destroy()` returns.



A side effect of this behavior is that a thread that's created by the thread pool can't call `thread_pool_destroy()` because the thread pool count will never drop to zero, and subsequently the function will never return.

Returns:**0**

Success.

-1

An error occurred.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>

int main( int argc, char **argv ) {
    thread_pool_t    *tpp;

    ...

    thread_pool_destroy ( tpp );
}
```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

thread_pool_limits()

Convenience wrapper function for *thread_pool_control()*

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int thread_pool_limits( thread_pool_t * pool,
                        int lowater,
                        int hiwater,
                        int maximum,
                        int increment,
                        unsigned flags );
```

Arguments:

pool

A thread pool handle that was returned by [thread_pool_create\(\)](#) (p. 3387).

lowater

The minimum number of threads that the pool should keep in the blocked state (i.e. threads that are ready to do work), or a negative number if you don't want to change the current value.

hiwater

The maximum number of threads that the pool should keep in the blocked state, or a negative number if you don't want to change the current value.

maximum

The maximum number of threads that the pool can create, or a negative number if you don't want to change the current value.

increment

The number of new threads created at one time, or a negative number if you don't want to change the current value.

flags

The only flag that's accepted is [THREAD_POOL_CONTROL_NONBLOCK](#) (p. 3386). For more information, see the documentation for *thread_pool_control()*.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `thread_pool_limits()` function is a wrapper function for [thread_pool_control\(\)](#) (p. 3385). If the value of *lowater*, *hiwater*, *maximum* or *increment* is 0 then that value is adjusted in the thread pool according to the handle *pool*.

If you don't set `THREAD_POOL_CONTROL_NONBLOCK`, the upper and lower bounds for waiting are:

- `lower = (lowater != -1) : lowater ? 0;`
- `upper = (maximum != -1) : maximum ? USHRT_MAX;`



Having several threads call this function with the same thread pool handle isn't recommended.

Returns:

-1 if an error occurs ([errno](#) (p. 539) is set).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

thread_pool_start()

Start a thread pool

Synopsis:

```
#include <sys/dispatch.h>

int thread_pool_start( void *pool );
```

Arguments:

pool

A thread pool handle that was returned by [thread_pool_create\(\)](#) (p. 3387).

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *thread_pool_start()* function starts the thread pool *pool*. The function may or may not return, depending on the flags that you passed to [thread_pool_create\(\)](#) (p. 3387).

Returns:

EOK

Success.

-1

An error occurred.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    thread_pool_attr_t    pool_attr;
    thread_pool_t         *tpp;
    dispatch_t            *dpp;
    resmgr_attr_t         attr;
    resmgr_context_t      *ctp;

    if( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
```

```

        dispatch context.\n", argv[0] );
    return EXIT_FAILURE;
}

memset( &pool_attr, 0, sizeof (pool_attr) );
pool_attr.handle = dpp;
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.unblock_func = dispatch_unblock;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

if( (tpp = thread_pool_create( &pool_attr,
    POOL_FLAG_EXIT_SELF )) == NULL ) {
    fprintf( stderr, "%s: Unable to initialize \
        thread pool.\n", argv[0] );
    return EXIT_FAILURE;
}

...

/* Never returns */
thread_pool_start( tpp );
}

```

For examples using the dispatch interface, see [dispatch_create\(\)](#) (p. 433), [message_attach\(\)](#) (p. 1624), [resmgr_attach\(\)](#) (p. 2580), and [thread_pool_create\(\)](#) (p. 3387).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

ThreadCancel(), ThreadCancel_r()

Cancel a thread

Synopsis:

```
#include <sys/neutrino.h>

int ThreadCancel( int tid,
                  void (*canstub)(void) );

int ThreadCancel_r( int tid,
                    void (*canstub)(void) );
```

Arguments:

tid

The ID of the thread that you want to destroy, as returned by [ThreadCreate\(\)](#) (p. 3402).

canstub

A pointer to the location that you want the thread to jump to when the cancellation occurs; see below.



You must provide a *canstub* function.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls request that the thread specified by *tid* be canceled. The target thread's cancelability state and type determine when the cancellation takes effect.

The *ThreadCancel()* and *ThreadCancel_r()* functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 3401) section for details.



Instead of using these kernel calls directly, consider calling [pthread_cancel\(\)](#) (p. 2251).

When the cancellation is acted upon, the thread jumps to the location specified by *canstub*. This stub should call cancellation cleanup handlers for the thread. When the last cancellation cleanup handler returns, the stub must terminate the thread using:

```
ThreadDestroy( 0, -1, PTHREAD_CANCEL);
```

Unlike [ThreadDestroy\(\)](#) (p. 3417), which destroys a thread immediately, [ThreadCancel\(\)](#) requests that the target thread execute any cleanup code and then terminate at its earliest convenience.

The cancellation processing in the target thread runs asynchronously with respect to the calling thread, which doesn't block.

The combinations of cancelability state and type are as follows:

State	Type	Description
Disabled	Deferred	Cancel requests are made pending
Disabled	Async	Cancel requests are made pending
Enabled	Deferred	Cancellation happens at the next cancellation point. These are at explicitly coded calls to pthread_testcancel() (p. 2432) or an attempt to enter a blocking state in any of the calls defined in the table below. All kernel calls that block are cancellation points, with the exception of MsgSendvnc() (p. 1805) and SyncMutexLock() (p. 3294).
Enabled	Async	Cancellation happens immediately.

Use [pthread_setcancelstate\(\)](#) (p. 2392), and [pthread_setcanceltype\(\)](#) (p. 2394) to set the state and type.

POSIX defines a list of functions that are cancellation points; some functions that aren't listed there may also be cancellation points. Any function that calls a blocking kernel call that's a cancellation point will itself become a cancellation point when the kernel call is made. The most common blocking kernel call in library code is [MsgSendv\(\)](#) (p. 1801).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ThreadCancel()

If an error occurs, the function returns -1 and sets *errno* (p. 539). Any other value returned indicates success.

ThreadCancel_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:**ESRCH**

The thread indicated by *tid* doesn't exist.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ThreadCreate(), ThreadCreate_r()

Create a thread

Synopsis:

```
#include <sys/neutrino.h>

int ThreadCreate(
    pid_t pid,
    void* (func)( void* ),
    void* arg,
    const struct _thread_attr* attr );

int ThreadCreate_r(
    pid_t pid,
    void* (func)( void* ),
    void* arg,
    const struct _thread_attr* attr );
```

Arguments:

pid

The ID of the process that you want to create the thread in, or 0 to create the thread in the current process.

func

A pointer to the function that you want the thread to execute. The *arg* argument that you pass to *ThreadCreate()* is passed to *func()* as its sole argument. If *func()* returns, it returns to the address defined in the *__exitfunc* member of *attr*.

arg

A pointer to any data that you want to pass to *func*.

attr

A pointer to a *_thread_attr* structure that specifies the attributes for the new thread, or NULL if you want to use the default attributes. For more information, see “[Thread attributes](#) (p. 3403),” below.



If you modify the attributes after creating the thread, the thread isn't affected.

Library:**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls create a new thread of execution, with attributes specified by *attr*, within the process specified by *pid*. If *pid* is zero, the current process is used. The *ThreadCreate()* and *ThreadCreate_r()* functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 3405) section for details.



- Only the Process Manager can create threads in another process.
- Instead of using these kernel calls directly, consider calling [pthread_create\(\)](#) (p. 2284).

The new thread shares all resources of the process in which it's created. This includes memory, timers, channels and connections. The standard C library contains mutexes to make it thread-safe.

Thread attributes

The `_thread_attr` structure pointed to by *attr* contains at least the following members:

int __flags

Zero or more of the following (the default of each pair of flags is always zero):

- `PTHREAD_CREATE_JOINABLE` (default) — put the thread into a zombie state when it terminates. It stays in this state until you retrieve its exit status or detach the thread.
- `PTHREAD_CREATE_DETACHED` — create the thread in the detached state; it doesn't become a zombie. You can't call [ThreadJoin\(\)](#) (p. 3422) for a detached thread.
- `PTHREAD_INHERIT_SCHED` (default) — use the scheduling attributes of the creating thread for the new thread.
- `PTHREAD_EXPLICIT_SCHED` — take the scheduling policy and parameters for the new thread from the `__policy` and `__param` members of *attr*.
- `PTHREAD_SCOPE_SYSTEM` (default) — schedule the thread against all threads in the system.
- `PTHREAD_SCOPE_PROCESS` — don't set this flag; the QNX Neutrino RTOS implements true microkernel threads that have only a system scope.

- `PTHREAD_MULTISIG_ALLOW` (default) — if the thread dies because of an unblocked, uncaught signal, terminate all threads, and hence, the process.
- `PTHREAD_MULTISIG_DISALLOW` — terminate only this thread; all other threads in the process are unaffected.
- `PTHREAD_CANCEL_DEFERRED` (default) — cancellation occurs only at cancellation points as defined by [ThreadCancel\(\)](#) (p. 3399).
- `PTHREAD_CANCEL_ASYNCHRONOUS` — every opcode executed by the thread is considered a cancellation point. The POSIX and C library aren't asynchronous-cancel safe.

size_t __stacksize

If `__stackaddr` isn't `NULL`, then `__stacksize` specifies the size of the thread stack defined in `__stackaddr`. If the stack size is less than `PTHREAD_STACK_MIN`, the call fails with an error of `EINVAL`.

If `__stackaddr` is `NULL`, then `__stacksize` specifies the minimum size of stack to dynamically allocate. If you set `__stacksize` to zero, then a system-defined amount is used.

void* __stackaddr

`NULL`, or the address of a stack that you want the thread to use. Set the `__stacksize` member to the size of the stack.

If you provide a non-`NULL` `__stackaddr`, it's your responsibility to release the stack when the thread dies. If `__stackaddr` is `NULL`, then the kernel dynamically allocates a stack on thread creation and automatically releases it on the thread's death.

void* (__exitfunc)(void* status)

The address to return to if the thread function returns.

The thread *returns* to `__exitfunc`. This means that the `status` variable isn't passed as a normal parameter. Instead, it appears in the return-value position dictated by the CPU's calling convention (e.g. `EAX` on an x86, and so on).



The `__exitfunc` function normally has to have compiler- and CPU-specific manipulation to access the `status` data (pulling it from the return register location to a proper local variable). Alternatively, you can write the `__exitfunc` function in assembly language for each CPU.

int __policy

The scheduling policy, as defined by the [SchedSet\(\)](#) (p. 2780) kernel call. This member is used only if you set the PTHREAD_EXPLICIT_SCHED flag. If you want the thread to inherit the policy, but you want to specify the scheduling parameters in the __param member, set the PTHREAD_EXPLICIT_SCHED flag and set the __policy member to SCHED_NOCHANGE.

struct sched_param __param

A [sched_param](#) (p. 2731) structure that specifies the scheduling parameters, as defined by the [SchedSet\(\)](#) kernel call. This member is used only if you set the PTHREAD_EXPLICIT_SCHED flag.

unsigned __guardsize

The amount of memory to use for the guard area of the thread's stack; for more information, see [pthread_attr_setguardsize\(\)](#) (p. 2217)

unsigned __prealloc

The amount of the memory to preallocate for a thread's MAP_LAZY stack; for more information, see [pthread_attr_setstackprealloc\(\)](#) (p. 2233).

Signal state

The signal state of the new thread is initialized as follows:

- The signal mask is inherited from the creating thread.
- The set of pending signals is empty.
- The cancel state and type are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED.

Local storage for private data

Each thread contains a thread local storage area for its private data. User applications don't need to directly access this area, but you can get a pointer to it by calling [__tls\(\)](#). The thread local storage is defined by the structure `_thread_local_storage`. Both of these are defined in `<sys/storage.h>`.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ThreadCreate()

The thread ID of the newly created thread. If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

ThreadCreate_r()

The thread ID of the newly created thread. This function does **NOT** set *errno*. If an error occurs, the function returns the negative of a value from the Errors section.

Errors:

EAGAIN

All kernel thread objects are in use.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The scheduling policy or priority was invalid, or the stack size is less than PTHREAD_STACK_MIN.

ENOTSUP

PTHREAD_SCOPE_PROCESS was requested. All kernel threads are PTHREAD_SCOPE_SYSTEM.

EPERM

The calling thread doesn't have sufficient permission to create a thread in another process. Only a thread with a process ID of 1 can create threads in other processes.

ESRCH

The process indicated by *pid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The QNX Neutrino interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```
void nothingthread( void )  
{  
    return;  
}
```

ThreadCtl(), ThreadCtl_r()

Control a thread

Synopsis:

```
#include <sys/neutrino.h>

int ThreadCtl( int cmd,
               void * data );

int ThreadCtl_r( int cmd,
                 void * data );
```

Arguments:

cmd

The command you want to execute; one of the following:

- [_NTO_TCTL_ALIGN_FAULT](#) (p. 3409)
- [_NTO_TCTL_IO](#) (p. 3409)
- [_NTO_TCTL_IO_PRIV](#) (p. 3410)
- [_NTO_TCTL_NAME](#) (p. 3410)
- [_NTO_TCTL_ONE_THREAD_CONT](#) (p. 3412)
- [_NTO_TCTL_ONE_THREAD_HOLD](#) (p. 3412)
- [_NTO_TCTL_RCM_GET_AND_SET](#) (p. 3412)
- [_NTO_TCTL_RUNMASK](#) (p. 3412)
- [_NTO_TCTL_RUNMASK_GET_AND_SET](#) (p. 3413)
- [_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT](#) (p. 3413)
- [_NTO_TCTL_THREADS_CONT](#) (p. 3415)
- [_NTO_TCTL_THREADS_HOLD](#) (p. 3415)

For more information, see below.

data

A pointer to data associated with the specific command; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls allow you to make OS-specific changes to a thread.

The *ThreadCtl()* and *ThreadCtl_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3415) section for details.

The sections that follow describe the possible commands.

_NTO_TCTL_ALIGN_FAULT

ThreadCtl(_NTO_TCTL_ALIGN_FAULT, *data*)

This command controls the response to a misaligned access. The *data* argument must be a pointer to an `int` whose value indicates how you want to respond:

- greater than 0 — make a misaligned access fault with a **SIGBUS**, if the architecture permits it.
- less than 0 — make the kernel attempt to emulate an instruction with a misaligned access. If the attempt fails, it also faults with a **SIGBUS**.
- 0 — don't change the alignment-fault handling for the thread.

The function sets *data* to a positive or negative number, indicating the previous state of the alignment-fault handling.

-
- Threads created by the calling thread inherit the `_NTO_TCTL_ALIGN_FAULT` status.
 - In QNX Neutrino 6.6 and later, on ARMv7 targets, the `_NTO_TCTL_ALIGN_FAULT` command is ignored; you can't control the alignment-fault behavior on a per-thread basis on these targets. You can set the global behavior by specifying the `-ad` or `-ae` option to `procnto*`:



-ad

Causes alignment faults for all threads (the default).

-ae

Performs hardware misaligned accesses and causes alignment faults only for certain misaligned accesses that can't be performed by hardware.

_NTO_TCTL_IO

ThreadCtl(_NTO_TCTL_IO, 0)

Superlock the process's memory and request I/O privileges; let the thread execute the `in`, `ins`, `out`, `outs`, `cli`, and `sti` I/O opcodes on architectures where it has the appropriate privilege, and let it attach IRQ handlers. If a thread attempts to use these

opcodes without successfully executing this call, the thread faults with a SIGSEGV when the opcode is attempted. A lack of I/O privileges may also cause a SIGILL signal.



- In order to use this command, your process must have the PROCmgr_AID_IO ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- Threads created by the calling thread inherit the _NTO_TCTL_IO status.

_NTO_TCTL_IO_PRIV

ThreadCtl(_NTO_TCTL_IO_PRIV, 0)

This command is similar to [_NTO_TCTL_IO](#) (p. 3409), but _NTO_TCTL_IO_PRIV also requests that the thread be put into System (privileged) execution mode on ARM targets.



- In order to use this command, your process must have the PROCmgr_AID_IO ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- Threads created by the calling thread inherit the _NTO_TCTL_IO_PRIV status.

_NTO_TCTL_NAME

ThreadCtl(_NTO_TCTL_NAME, *data*)

(QNX Neutrino Core OS 6.3.2 or later) Set or retrieve the name of the current thread. The *data* argument must be a pointer to a `_thread_name` structure, which is defined as follows:

```
struct _thread_name {
    int    new_name_len;
    int    name_buf_len;
    char   name_buf[1];
};
```

The *name_buf* member is a contiguous buffer that extends the structure; *name_buf_len* is the size of this buffer.

If <i>new_name_len</i> is:	The command:
Less than 0	Copies the thread's current name into <i>name_buf</i> , up to the number of bytes specified by <i>name_buf_len</i>
0	Deletes the thread's name

If <i>new_name_len</i> is:	The command:
Greater than 0	Sets the thread's name to the value in <i>name_buf</i> , up to the number of bytes specified by <i>name_buf_len</i>

If you're setting or deleting the thread's name, the old name is copied as a NULL-terminated string into *name_buf*, up to the number of bytes specified by *name_buf_len*.



- Currently, the thread names are limited to `_NTO_THREAD_NAME_MAX`.
- You can also use the [pthread_getname_np\(\)](#) (p. 2297) and [pthread_setname_np\(\)](#) (p. 2398) functions instead of calling `ThreadCtl()` directly.

Here's an example:

```
#include <stdio.h>
#include <sys/neutrino.h>
#include <stdlib.h>

int main () {
    struct _thread_name *tname;
    int size;

    size = sizeof(*tname) * 2 + _NTO_THREAD_NAME_MAX * sizeof(char);
    tname = malloc (size);

    if (tname == NULL) {
        perror ("malloc");
        return EXIT_FAILURE;
    } else {
        memset (tname, 0x00, size);
        tname->name_buf_len = _NTO_THREAD_NAME_MAX;

        /* To change the name, put the name into name_buf and set
           new_name_len to the length of the new name. */

        strcpy (tname->name_buf, "Hello!");
        tname->new_name_len = strlen (tname->name_buf);

        if (ThreadCtl (_NTO_TCTL_NAME, tname) == -1) {
            perror ("ThreadCtl()");
            return EXIT_FAILURE;
        } else {
            printf ("The old name was: '%s'.\n", tname->name_buf);
        }

        /* To get the current name, set new_name_len to -1. */

        tname->new_name_len = -1;

        if (ThreadCtl (_NTO_TCTL_NAME, tname) == -1) {
            perror ("ThreadCtl()");
            return EXIT_FAILURE;
        } else {
            printf ("The current name is: '%s'.\n",
                    tname->name_buf);
        }

        /* To delete the name, set new_name_len to 0. */

        tname->new_name_len = 0;

        if (ThreadCtl (_NTO_TCTL_NAME, tname) == -1) {
```

```
        perror ("ThreadCtl()");
        return EXIT_FAILURE;
    } else {
        printf ("The old name was: '%s'.\n", tname->name_buf);
    }
    free (tname);
}
return EXIT_SUCCESS;
}
```

`_NTO_TCTL_ONE_THREAD_CONT`

ThreadCtl(`_NTO_TCTL_ONE_THREAD_CONT`, *data*)

(QNX Neutrino Core OS 6.3.2 or later) Unfreeze the thread with the given thread ID, which was frozen by an earlier `_NTO_TCTL_ONE_THREAD_HOLD` command. The *data* is the thread ID, cast to be a pointer (i.e. `(void *) tid`). This command returns an error of `ESRCH` if there's no thread with an ID of *tid*.

`_NTO_TCTL_ONE_THREAD_HOLD`

ThreadCtl(`_NTO_TCTL_ONE_THREAD_HOLD`, *data*)

(QNX Neutrino Core OS 6.3.2 or later) Hold the thread with the given thread ID in the calling process. The *data* is the thread ID, cast to be a pointer (i.e. `(void *) tid`). This command returns an error of `ESRCH` if there's no thread with an ID of *tid*.

`_NTO_TCTL_RCM_GET_AND_SET`

ThreadCtl(`_NTO_TCTL_RCM_GET_AND_SET`, *data*)

(QNX Neutrino 6.6 or later) Enter or leave resource-constrained mode. The *data* is a pointer to an integer; set the integer to 1 to entry constrained mode, or to 0 to leave it. The call stores the previous value in the memory pointed to by *data*:

```
int value = 1;
ThreadCtl(_NTO_TCTL_RCM_GET_AND_SET, &value); /* swaps current state with value */
/* Handle the request... */
ThreadCtl(_NTO_TCTL_RCM_GET_AND_SET, &value); /* restores original state */
```

For more information, see Resource constraint thresholds in the Processes chapter of the QNX Neutrino *Programmer's Guide*.

`_NTO_TCTL_RUNMASK`

ThreadCtl(`_NTO_TCTL_RUNMASK`, *data*)

Set the processor affinity for the calling thread in a multiprocessor system. The *data* is the runmask, cast to be a pointer (i.e. `(void *) runmask`). Each set bit in *runmask* represents a processor that the thread can run on.

By default, a thread's *runmask* is set to all ones, which allows it to run on any available processor. A value of `0x01` would, for example, force the thread to run only on the first processor.

You can use `_NTO_TCTL_RUNMASK` to optimize the runtime performance of your system by, for example, relegating nonrealtime threads to a specific processor. In

general, this shouldn't be necessary, since the QNX Neutrino realtime scheduler always preempts a lower-priority thread immediately when a higher priority thread becomes ready.

The main effect of processor locking is the effectiveness of the CPU cache, since threads can be prevented from migrating.



Threads created by the calling thread don't inherit the specified runmask.

_NTO_TCTL_RUNMASK_GET_AND_SET

ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET, *data*)

(QNX Neutrino 6.3.0 SP2 or later) Get and set the runmask (the processor affinity) to a proper value for the calling thread in a multiprocessor system. The *data* parameter is a pointer to the runmask. On input, the pointer to value is used to set the new runmask for the thread (see `_NTO_TCTL_SET_RUNMASK` for details). After the function has completed, the contents of **data* will be replaced with the previous runmask for the thread. Calling *ThreadCtl* again with the same pointer will restore the runmask to the state before the call.

_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT

ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT, *data*)

(QNX Neutrino Core OS 6.3.2 or later) Manipulate the calling thread's runmask and inherit mask. The *data* argument must be a pointer to a struct `_thread_runmask`. Conceptually, this structure consists of these members:

- *size*
- *runmask*
- *inherit_mask*

However, the size of the masks (and hence the size of the structure) depends on the number of processors on your system. We've defined the following macros to make it easier for you to work with this structure:

RMSK_SIZE(num_cpu)

Determine the size of the masks. You can find the number of CPUs on your system in `_syspage_ptr->num_cpu`.

RMSK_SET(cpu, p)

Set the bit for *cpu* (where *cpu* is zero-based) in the mask *p*.

RMSK_CLR(cpu, p)

Clear the bit for *cpu* (where *cpu* is zero-based) in the mask *p*.

RMSK_ISSET(cpu, p)

Determine the value of the bit for *cpu* in the mask *p*.

The `_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT` command saves the values for both masks at the time of the call in their respective members of this structure. If you pass 0 for the masks, the masks are left unaltered; otherwise they're set to the specified value(s).

Here's an example:

```
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <malloc.h>
#include <stdio.h>

int
main(void)
{
    int *rsizep, rsize, size_tot;
    unsigned *rmaskp, *inheritp;
    unsigned buf[8];
    void *freep;

    /*
     * struct _thread_runmask is not
     * uniquely sized, so we construct
     * our own.
     */

    rsize = RMSK_SIZE(_syspage_ptr->num_cpu);

    size_tot = sizeof(*rsizep);
    size_tot += sizeof(*rmaskp) * rsize;
    size_tot += sizeof(*inheritp) * rsize;

    if (size_tot <= sizeof(buf)) {
        rsizep = buf;
        freep = NULL;
    }
    else if ((rsizep = freep = malloc(size_tot)) == NULL) {
        perror("malloc");
        return 1;
    }

    memset(rsizep, 0x00, size_tot);

    *rsizep = rsize;
    rmaskp = (unsigned *) (rsizep + 1);
    inheritp = rmaskp + rsize;

    /*
     * Both masks set to 0 means get the current
     * values without alteration.
     */

    if (ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT,
                  rsizep) == -1) {
        perror("_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT");
        free(freep);
        return 1;
    }

    /*
     * Restrict our inherit mask to the last cpu; leave the
     * runmask unaltered.
     */

    memset(rsizep, 0x00, size_tot);
    *rsizep = rsize;
    RMSK_SET(_syspage_ptr->num_cpu - 1, inheritp);

    if (ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT,
```

```

        rsizep) == -1) {
    perror("_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT");
    free(freep);
    return 1;
}

free(freep);
return 0;
}

```

_NTO_TCTL_THREADS_CONT

ThreadCtl(_NTO_TCTL_THREADS_CONT, 0)

Unfreeze all threads in the current process that were frozen using the _NTO_TCTL_THREADS_HOLD command.

_NTO_TCTL_THREADS_HOLD

ThreadCtl(_NTO_TCTL_THREADS_HOLD, 0)

Hold all threads in the current process except the calling thread.



Threads created by the calling thread aren't frozen.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

***ThreadCtl*()**

If an error occurs, this function returns -1 and sets *errno* (p. 539). Any other value returned indicates success.

***ThreadCtl_r*()**

This function returns EOK on success. It **doesn't** set *errno*. If an error occurs, the function may return any value listed in the Errors section.

Errors:

E2BIG

The name is larger than the accepted size.

EINVAL

The name buffer length is invalid or smaller than the new name length, or the specified runmask is invalid.

EPERM

The calling process doesn't have the required permission; if you're using `_NTO_TCTL_IO`, see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

There's no thread with an ID of *tid*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ThreadDestroy(), ThreadDestroy_r()

Destroy a thread immediately

Synopsis:

```
#include <sys/neutrino.h>

int ThreadDestroy( int tid,
                  int priority,
                  void* status );

int ThreadDestroy_r( int tid,
                    int priority,
                    void* status );
```

Arguments:

tid

The ID of the thread that you want to destroy, as returned by [ThreadCreate\(\)](#) (p. 3402), or 0 to destroy the current thread, or -1 to destroy all the threads in the current process.

priority

The priority at which you want to destroy multiple threads, or -1 to use the priority of the current thread.

status

The value to make available to a thread that joins a nondetached thread that's destroyed.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

These kernel calls terminate the thread specified by *tid*. If *tid* is 0, the calling thread is assumed. If *tid* is -1, all of the threads in the process are destroyed. When multiple threads are destroyed, the destruction is scheduled one thread at a time at the priority specified by the *priority* argument. If *priority* is -1, then the priority of the calling thread is used.

The *ThreadDestroy()* and *ThreadDestroy_r()* functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 3418) section for details.



Instead of using these kernel calls directly, consider calling [pthread_abort\(\)](#) (p. 2185) or [pthread_exit\(\)](#) (p. 2292)

If a terminated thread isn't detached, it makes the value specified by the *status* argument available to any successful join on it. Until another thread retrieves this value, the thread ID *tid* isn't reused, and a small kernel resource (a thread object) is held in the system. If the thread is detached, then *status* is ignored, and all thread resources are immediately released.

When the last thread in a process is destroyed, the process terminates, and all thread resources are released, even if they're not detached and unjoined.



On return from *ThreadDestroy()* or *ThreadDestroy_r()*, the target thread is marked for death, but if it isn't possible to kill it immediately, it may not be terminated until it attempts to run.

Blocking states

If these calls return, they don't block.

Returns:

If the calling thread is destroyed, *ThreadDestroy()* and *ThreadDestroy_r()* don't return.

The only difference between these functions is the way they indicate errors:

ThreadDestroy()

If this function returns and an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

ThreadDestroy_r()

EOK is returned on success. This function does **NOT** set *errno*. If this function returns and an error occurs, any value in the Errors section may be returned.

Errors:

ESRCH

The thread indicated by *tid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ThreadDetach(), ThreadDetach_r()

Detach a thread from a process

Synopsis:

```
#include <sys/neutrino.h>

int ThreadDetach( int tid );
int ThreadDetach_r( int tid );
```

Arguments:

tid

The ID of the thread that you want to detach, as returned by [ThreadCreate\(\)](#) (p. 3402), or 0 to detach the current thread.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls detach the thread specified by *tid*. If *tid* is zero, the calling thread is used. Once detached, attempts to call *ThreadJoin()* on *tid* fail. When a detached thread terminates, its termination status is discarded and all its resources are released.

The *ThreadDetach()* and *ThreadDetach_r()* functions are identical, except in the way they indicate errors. See the [Returns](#) (p. 3420) section for details.



Instead of using these kernel calls directly, consider calling [pthread_detach\(\)](#) (p. 2288).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

ThreadDetach()

If an error occurs, the function returns -1 and sets [errno](#) (p. 539). Any other value returned indicates success.

ThreadDetach_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function can return any value listed in the Errors section.

Errors:**EINVAL**

The thread is already detached.

ESRCH

The thread indicated by *tid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ThreadJoin(), ThreadJoin_r()

Block until a thread terminates

Synopsis:

```
#include <sys/neutrino.h>

int ThreadJoin( int tid,
                void** status );

int ThreadJoin_r( int tid,
                  void** status );
```

Arguments:

tid

The ID of the thread that you want to detach, as returned by [ThreadCreate\(\)](#) (p. 3402).

status

The address of a pointer to a location where the function can store the thread's exit status.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ThreadJoin()* and *ThreadJoin_r()* kernel calls block until the thread specified by *tid* terminates. If *status* isn't NULL, the functions save the thread's exit status in the area pointed to by *status*. If the thread *tid* has already terminated, the functions immediately return with success and the status, if requested.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3423) section for details.



Instead of using these kernel calls directly, consider calling [pthread_join\(\)](#) (p. 2303) or [pthread_timedjoin\(\)](#) (p. 2433).

When *ThreadJoin()* returns successfully, the target thread has been successfully terminated. Until this occurs, the thread ID *tid* isn't reused and a small kernel resource (a thread object) is retained.

You can't join a thread that's detached (see *ThreadCreate()* (p. 3402) and *ThreadDetach()* (p. 3420)).

The target thread must be joinable. Multiple *pthread_join()* (p. 2303), *pthread_timedjoin()* (p. 2433), *ThreadJoin()*, and *ThreadJoin_r()* calls on the same target thread aren't allowed.

Blocking states

STATE_JOIN

The calling thread blocks waiting for the indicated thread to exit.

Returns:

The only difference between these functions is the way they indicate errors:

ThreadJoin()

If an error occurs, the function returns -1 and sets *errno* (p. 539). Any other value returned indicates success.

ThreadJoin_r()

Returns EOK on success. This function does **NOT** set *errno*. If an error occurs, the function may return any value listed in the Errors section.

Errors:

EBUSY

Attempt to join a thread which has been joined by another thread.

EDEADLK

Attempt to join to yourself.

EFAULT

A fault occurred when the kernel tried to access *status*.

EINTR

The call was interrupted by a signal.

EINVAL

Attempt to join a thread which is detached (see *ThreadDetach()*).

ESRCH

The thread indicated by *tid* doesn't exist.

ETIMEDOUT

A kernel timeout unblocked the call. See [TimerTimeout\(\)](#) (p. 3464).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

time()

Determine the current calendar time

Synopsis:

```
#include <time.h>

time_t time( time_t * tloc );
```

Arguments:

tloc

NULL, or a pointer to a `time_t` object where the function can store the current calendar time.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `time()` function takes a pointer to `time_t` as an argument and returns a value of `time_t` on exit. The returned value is the current calendar time, in seconds, since the Unix Epoch, 00:00:00 January 1, 1970 Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

You typically use the `date` command to set the computer's internal clock using Coordinated Universal Time (UTC). Use the **TZ** environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Returns:

The current calendar time, in seconds, since 00:00:00 January 1, 1970 Coordinated Universal Time (UTC). If *tloc* isn't NULL, the current calendar time is also stored in the object pointed to by *tloc*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>

int main( void )
{
    time_t time_of_day;

    time_of_day = time( NULL );
    printf( "It is now: %s", ctime( &time_of_day ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
It is now: Wed Jun 30 09:09:33 1999
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timer_create()

Create a timer

Synopsis:

```
#include <signal.h>
#include <time.h>

int timer_create( clockid_t clock_id,
                  struct sigevent * evp,
                  timer_t * timerid );
```

Arguments:

clock_id

The clock source that you want to use; one of:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

evp

NULL, or a pointer to a [sigevent](#) (p. 2961) structure containing the event that you want to deliver when the timer fires.

timerid

A pointer to a `timer_t` object where the function stores the ID of the new timer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_create()` function creates a per-process timer using the specified clock source, `clock_id`, as the timing base.



- This function fails if the clock ID corresponds to the CPU-time clock of a process or thread different from the process or thread invoking the function.
 - In order to create a timer that sends a pulse, your process must have the `PROCmgr_AID_TIMER` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
-

You can use the time ID that the function stores in `timerid` in subsequent calls to [timer_gettime\(\)](#) (p. 3437), [timer_settime\(\)](#) (p. 3439), and [timer_delete\(\)](#) (p. 3431).

The timer is created in the disabled state, and isn't enabled until you call `timer_settime()`.

We recommend the following event types:

- `SIGEV_SIGNAL`
- `SIGEV_SIGNAL_CODE`
- `SIGEV_SIGNAL_THREAD`
- `SIGEV_PULSE`

If the `evp` argument is `NULL`, a `SIGALRM` signal is sent to your process when the timer expires. To specify a handler for this signal, call [sigaction\(\)](#) (p. 2948).

Returns:

0

Success. The `timerid` argument is set to the timer's ID.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

All timers are in use. You'll have to wait for a process to release one.

EINVAL

The `clock_id` isn't one of the valid `CLOCK_*` constants.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

```

/*
 * Demonstrate how to set up a timer that, on expiry,
 * sends us a pulse. This example sets the first
 * expiry to 1.5 seconds and the repetition interval
 * to 1.5 seconds.
 */

#include <stdio.h>
#include <time.h>
#include <sys/netmgr.h>
#include <sys/neutrino.h>

#define MY_PULSE_CODE    _PULSE_CODE_MINAVAIL

typedef union {
    struct _pulse    pulse;
    /* your other message structures would go
     here too */
} my_message_t;

main()
{
    struct sigevent    event;
    struct itimerspec    itime;
    timer_t            timer_id;
    int                chid;
    int                rcvid;
    my_message_t        msg;

    chid = ChannelCreate(0);

    event.sigev_notify = SIGEV_PULSE;
    event.sigev_coid = ConnectAttach(ND_LOCAL_NODE, 0,
                                    chid,
                                    _NTO_SIDE_CHANNEL, 0);

    event.sigev_priority = getprio(0);
    event.sigev_code = MY_PULSE_CODE;
    timer_create(CLOCK_REALTIME, &event, &timer_id);

    itime.it_value.tv_sec = 1;
    /* 500 million nsecs = .5 secs */
    itime.it_value.tv_nsec = 500000000;
    itime.it_interval.tv_sec = 1;
    /* 500 million nsecs = .5 secs */
    itime.it_interval.tv_nsec = 500000000;
    timer_settime(timer_id, 0, &itime, NULL);

    /*
     * As of the timer_settime(), we will receive our pulse
     * in 1.5 seconds (the itime.it_value) and every 1.5
     * seconds thereafter (the itime.it_interval)
     */

    for (;;) {
        rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
        if (rcvid == 0) { /* we got a pulse */
            if (msg.pulse.code == MY_PULSE_CODE) {
                printf("we got a pulse from our timer\n");
            } /* else other pulses ... */
        } /* else other messages ... */
    }
}

```

Classification:

POSIX 1003.1 TMR (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This version of *timer_create()* is different from the QNX 4 version, which was based on a draft standard.

timer_delete()

Delete a timer

Synopsis:

```
#include <time.h>

int timer_delete( timer_t timerid );
```

Arguments:

timerid

A `timer_t` object that holds a timer ID, as set by [timer_create\(\)](#) (p. 3427).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_delete()` function removes a previously attached timer based upon the *timerid* returned from the `timer_create()` function. The timer is removed from the active system timer list, and returned to the free list of available timers.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EINVAL

The timer *timerid* isn't attached to the calling process.

Classification:

[POSIX 1003.1 TMR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timer_getexpstatus()

Get the expiry status of a timer

Synopsis:

```
#include <time.h>

int timer_getexpstatus( timer_t timerid );
```

Arguments:

timerid

A timer_t object that holds a timer ID, as set by [timer_create\(\)](#) (p. 3427).

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *timer_getexpstatus()* function gets the expiry status of the time with the ID given by *timerid*.



- This function is based on a POSIX draft. If you want your code to be more portable, then instead of calling *timer_getexpstatus()*, call [timer_gettime\(\)](#) (p. 3437) and check the amount of time left on the timer.
- In order to get timer information, your process must have the PROCmgr_AID_TIMER ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Returns:

0

The timer has expired.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EINVAL**

The timer specified by *timerid* doesn't exist.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timer_getoverrun()

Return the number of timer overruns

Synopsis:

```
#include <signal.h>
#include <time.h>

int timer_getoverrun( timer_t timerid );
```

Arguments:

timerid

A `timer_t` object that holds a timer ID, as set by [timer_create\(\)](#) (p. 3427).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

When a timer expiration signal is received by a process, the `timer_getoverrun()` function returns the timer expiration overrun count for the timer specified by `timerid`.

Only a single signal is queued to the process for a given timer at any point in time. When a timer that has a signal pending expires, no signal is queued and a timer overrun occurs.



- Pulses are queued for a process, so timer overruns occur only for signals.
- In order to get timer information, your process must have the `PROCmgr_AID_TIMER` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The overrun count returned is the number of extra timer expirations that occurred between the time the expiration signal was queued and when it was delivered or accepted, up to but not including `DELAYTIMER_MAX`. If the number of overruns is greater than or equal to `DELAYTIMER_MAX`, the overrun count is set to `DELAYTIMER_MAX`.

The value returned by *timer_getoverrun()* applies to the most recent expiration signal for the specified timer. If no expiration signal has been delivered, the overrun count is 0.

Returns:

The number of overruns, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:**EINVAL**

Invalid timer *timerid*.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 TMR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timer_gettime()

Get the amount of time left on a timer

Synopsis:

```
#include <time.h>

int timer_gettime( timer_t timerid,
                  struct itimerspec *value );
```

Arguments:

timerid

A `timer_t` object that holds a timer ID, as set by [timer_create\(\)](#) (p. 3427).

value

A pointer to a `itimerspec` structure that the function fills in with the timer's time until expiry. The structure contains at least the following members:

struct timespec it_value

A [timespec](#) (p. 3473) structure that contains the amount of time left before the timer expires, or zero if the timer is disarmed. This value is expressed as the relative interval until expiration, even if the timer was armed with an absolute time.

struct timespec it_interval

A `timespec` structure that contains the timer's reload value. If nonzero, it indicates a repetitive timer period.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_gettime()` function gets the amount of time left before the specified timer is to expire, along with the timer's reload value, and stores it in the space provided by the `value` argument.



In order to get timer information, your process must have the `PROCmgr_AID_TIMER` ability enabled. For more information, see [`procmgr\_ability\(\)`](#) (p. 2141).

Returns:

0

Success.

-1

An error occurred ([`errno`](#) (p. 539) is set).

Errors:**EINVAL**

The timer *timerid* isn't attached to the calling process.

EPERM

The calling process doesn't have the required permission; see [`procmgr\_ability\(\)`](#) (p. 2141).

Classification:

[POSIX 1003.1 TMR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timer_settime()

Set the expiration time for a timer

Synopsis:

```
#include <time.h>

int timer_settime( timer_t timerid,
                  int flags,
                  const struct itimerspec * value,
                  struct itimerspec * ovalue );
```

Arguments:

timerid

A `timer_t` object that holds a timer ID, as set by [timer_create\(\)](#) (p. 3427).

flags

The type of timer to arm if you aren't disarming the timer. The valid bits include:

- **TIMER_ABSTIME** — the *it_value* represents an absolute expiration date in seconds and nanoseconds from 1970. If the date specified has already passed, the function succeeds and the expiration notice is made.

If you don't set this bit, the *it_value* represents a relative expiration period that's offset from the current system time by the specified number of seconds and nanoseconds.

The following are QNX Neutrino extensions:

- **TIMER_TOLERANCE** — specify the amount of the tolerance to allow the kernel in low-power situations. You can't set this and **TIMER_ABSTIME** in the same call.
- **TIMER_PRECISE** — exclude the timer from timer harmonization. You can OR this into the flags if you're setting an expiration time (i.e., you didn't specify **TIMER_TOLERANCE**).

value

A pointer to a `itimerspec` structure that specifies the value that you want to set for the timer's time until expiry, or the timer tolerance, depending on the flags. For more information about this structure, see [timer_gettime\(\)](#) (p. 3437).

If you specify the `TIMER_TOLERANCE` flag, this argument can be `NULL`.

ovalue

`NULL`, or a pointer to a `itimerspec` structure that the function fills in with the timer's former time until expiry or tolerance, depending on the flags.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_settime()` function sets the expiration time of the timer specified by *timerid* from the *it_value* member of the *value* argument. If the *it_value* structure member of *value* is zero, then the timer is disarmed.

The `timer_settime()` function rounds up time values that are between two consecutive nonnegative integer multiples of the resolution of the specified timer to the larger multiple of the resolution.

If the *it_interval* member of *value* is nonzero, then it specifies a repeat rate that is added to the timer once the *it_value* period has expired. Subsequently, the timer is automatically rearmed, causing it to become continuous with a period of *it_interval*.

If the *ovalue* parameter isn't `NULL`, then on return from this function it contains a value representing the previous amount of time left before the timer was to have expired, or zero if the timer was disarmed. The previous interval timer period is also stored in the *it_interval* member.

The *timerid* is local to the calling process, and must have been created using `timer_create()`.



Because of the nature of time measurement, the timer might actually expire after the specified time. For more information, see the Tick, Tock:

Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Timer tolerance

As a QNX Neutrino extension, if you specify the `TIMER_TOLERANCE` flag, then:

- If *value* isn't `NULL`, the *it_value* member specifies the amount of tolerance, in nanoseconds, that the kernel is allowed in low-power situations. The kernel uses the expiry time plus the tolerance value to decide if it should enter tickless operation or a low-power mode.

- If *ovalue* isn't NULL, the function sets the *it_value* member to the previous tolerance.

You can set the tolerance at any time without affecting the active/inactive status of the timer. For more information, see “Clocks, timers, and power management” in the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EFAULT

A fault occurred trying to access the buffers provided.

EINVAL

The timer *timerid* isn't attached to the calling process or the number of nanoseconds specified by the *tv_nsec* member of one of the [timespec](#) (p. 3473) structures in the *itimerspec* structure pointed to by *value* is less than zero or greater than or equal to 1000 million.

Examples:

See [timer_create\(\)](#) (p. 3427).

Classification:

[POSIX 1003.1 TMR](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timer_timeout(), timer_timeout_r()

Set a timeout on a blocking state

Synopsis:

```
#include <time.h>

int timer_timeout(
    clockid_t id,
    int flags,
    const struct sigevent* notify,
    const struct timespec* ntime,
    struct timespec* otime );

int timer_timeout_r(
    clockid_t id,
    int flags,
    const struct sigevent* notify,
    const struct timespec* ntime,
    struct timespec* otime );
```

Arguments:

id

The clock to use to implement the timeout; one of:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

flags

A bitmask that specifies which states you want the timeout to apply to; see below.

You can also OR in at most one of the following bits:

- **TIMER_ABSTIME** — set an absolute expiration time.

- **TIMER_TOLERANCE** — specify the amount of the tolerance to allow the kernel in low-power situations.

If you don't set either of these bits, the function sets a relative expiration time.

If you're setting an expiration time (i.e., you didn't specify **TIMER_TOLERANCE**), you can OR in **TIMER_PRECISE** to exclude the timer from timer harmonization.

notify

A pointer to a [sigevent](#) (p. 2961) structure that defines the event that the kernel acts on if the timeout expires; see below.

ntime

A pointer to a [timespec](#) (p. 3473) structure that specifies the timeout.

otime

NULL, or a pointer to a location where the function can store the time remaining in the sleep.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_timeout()` and `timer_timeout_r()` functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3446) section for details.

The `timer_timeout()` function sets the timeout `ntime` on any kernel blocking state. The actual timeout that occurred is returned in `otime`. The resolution of `otime` for both `timer_timeout()` and [TimerTimeout\(\)](#) (p. 3464) functions is in nanoseconds. The difference, however, for `otime` in these two functions is in the format. For `timer_timeout()`, the `otime` is a pointer to `timespec` structure with two integers, whereas for `TimerTimeout()`, the pointer is of `uint64_t` type.

The time in `TimerTimeout()`'s `ntime` argument is also in nanoseconds. When `ntime` is passed to `TimerTimeout()`, the time (in `timespec`) is converted from seconds and nanoseconds into nanoseconds. When `otime` is returned to `timer_timeout()`, the time is converted from nanoseconds into seconds and nanoseconds.

The kernel blocking states are entered as a result of the following kernel calls:

Kernel function call	Blocking state
<i>InterruptWait()</i> (p. 1183)	STATE_INTR
<i>MsgReceivev()</i> (p. 1771)	STATE_RECEIVE
<i>MsgSendv()</i> (p. 1801)	STATE_SEND or STATE_REPLY
<i>SignalSuspend()</i> (p. 2995)	STATE_SIGSUSPEND
<i>SignalWaitinfo()</i> (p. 2997)	STATE_SIGWAITINFO
<i>SyncCondvarWait()</i> (p. 3282)	STATE_CONDVAR
<i>SyncMutexLock()</i> (p. 3294)	STATE_MUTEX
<i>SyncSemWait()</i> (p. 3303)	STATE_SEM
<i>ThreadJoin()</i> (p. 3422)	STATE_JOIN

The user specifies which states the timeout should apply to via a bitmask passed in the *flags* argument. The bits are defined by the following constants:

Constant	Meaning
_NTO_TIMEOUT_CONDVAR	Time out on STATE_CONDVAR.
_NTO_TIMEOUT_JOIN	Time out on STATE_JOIN.
_NTO_TIMEOUT_INTR	Time out on STATE_INTR.
_NTO_TIMEOUT_MUTEX	Time out on STATE_MUTEX.
_NTO_TIMEOUT_RECEIVE	Time out on STATE_RECEIVE.
_NTO_TIMEOUT_REPLY	Time out on STATE_REPLY.
_NTO_TIMEOUT_SEM	Time out on STATE_SEM.
_NTO_TIMEOUT_SEND	Time out on STATE_SEND.
_NTO_TIMEOUT_SIGSUSPEND	Time out on STATE_SIGSUSPEND.
_NTO_TIMEOUT_SIGWAITINFO	Time out on STATE_SIGWAITINFO.

For example, to set a timeout on *MsgSendv()*, specify:

```
_NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY
```

Once a timeout is specified using *timer_timeout()*, it's armed and released under the following conditions:

Armed

The kernel attempts to enter a blocking state specified in *flags*.

Released

One of the above kernel calls completed without blocking, *or* the kernel call blocks but unblocks before the timeout expires, *or* the timeout expires.

The *timer_timeout()* function always operates on a one-shot basis. When one of the above kernel calls returns (or is interrupted by a signal), the timeout request is removed from the system. Only one timeout per thread may be in effect at a time. A second call to *timer_timeout()*, without calling one of the above kernel functions, replaces the existing timeout on that thread. A call with *flags* set to zero ensures that a timeout won't occur on any state. This is the default when a thread is created.

Always call *timer_timeout()* just before the function that you wish to time out. For example:

```
...
event.sigev_notify = SIGEV_UNBLOCK;

timeout.tv_sec = 10;
timeout.tv_nsec = 0;

timer_timeout( CLOCK_REALTIME,
               _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY,
               &event, &timeout, NULL );
MsgSendv( coid, NULL, 0, NULL, 0 );
...
```

If the signal handler is called between the calls to *timer_timeout()* and *MsgSendv()* (p. 1801), the *timer_timeout()* values are saved during the signal handler and then are restored when the signal handler exits.

If the timeout expires, the kernel acts upon the event specified by the *sigevent* (p. 2961) structure pointed to by the *notify* argument. We recommend the following event types in this case:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE
- SIGEV_UNBLOCK
- SIGEV_INTR

Only SIGEV_UNBLOCK guarantees that the kernel call unblocks. A signal may be ignored, blocked, or accepted by another thread and a pulse can only unblock a *MsgReceivev()*. If a NULL is passed for *event* then SIGEV_UNBLOCK is assumed. In this case, a timed out kernel call will return failure with an error of ETIMEDOUT.



MsgSendv() (p. 1801) won't unblock on SIGEV_UNBLOCK if the server has already received the message via *MsgReceivev()* (p. 1771) and has specified _NTO_CHF_UNBLOCK in the *flags* argument to its *ChannelCreate()* (p. 278) call.

In this case, it's up to the server to do a [MsgReplyv\(\)](#) (p. 1778) or [MsgError\(\)](#) (p. 1743).

The timeout:

- is specified by the *ntime* argument
- is relative to the current time (when *timer_timeout()* is called), unless *flags* includes `TIMER_ABSTIME`, which makes the timeout occur at the absolute time set in *ntime*
- occurs on a clock tick (see [ClockPeriod\(\)](#) (p. 329) so the actual wakeup time is a minimum of:

$$(tv\_sec \times 1000000000 + tv\_nsec) / (\text{size of timer tick}) \text{ nanoseconds}$$

where *tv_sec* and *tv_nsec* are fields of the `timespec` structure (defined in `<time.h>`)



Because of the nature of time measurement, the timer might actually expire later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

If you don't wish to block for any time, you can pass a `NULL` for *ntime* in which case no timer is used, the event is assumed to be `SIGEV_UNBLOCK` and an attempt to enter a blocking state as set by *flags* will immediately return with `ETIMEDOUT`. Although a questionable practice, this can be used to poll potential blocking kernel calls. For example, you can poll for messages using *MsgReceivev()* with an immediate timeout. A much better approach is to use multiple threads and have one block waiting for messages.

If *flags* is set to `_NTO_TIMEOUT_NANOSLEEP`, then these calls block in the `STATE_NANOSLEEP` state until the timeout (or a signal which unblocks the thread) occurs. This can be used to implement an efficient kernel sleep as follows:

```
timer_timeout( CLOCK_REALTIME, _NTO_TIMEOUT_NANOSLEEP,  
              NULL, &ntime, &otime );
```

If *otime* isn't `NULL` and the sleep is unblocked by a signal then it contains the time remaining in the sleep.

Blocking states

The kernel calls don't block unless `_NTO_TIMEOUT_NANOSLEEP` is specified in *flags*. In this case, the calls block as follows:

STATE_NANOSLEEP

The calling thread blocks for the requested time period.

Returns:

timer_timeout()

The previous flags. If an error occurs, the function returns -1 and sets [errno](#) (p. 539).

timer_timeout_r()

The previous flags. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EAGAIN**

All kernel timer entries are in use.

EFAULT

A fault occurred when the kernel tried to access *ntime*, *otime*, or *notify*.

EINTR

The call was interrupted by a signal.

EINVAL

The clock type *id* isn't one of `CLOCK_MONOTONIC`, `CLOCK_SOFTTIME`, or `CLOCK_REALTIME`.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The timeout value *starts timing out* when *timer_timeout()* is called, *not* when the blocking state is entered. It might be possible to get preempted after calling *timer_timeout()* but before the blocking kernel call.

TimerAlarm(), TimerAlarm_r()

Send an alarm signal

Synopsis:

```
#include <sys/neutrino.h>

int TimerAlarm( clockid_t id,
                const struct _itimerval * itime,
                struct _itimerval * otime );

int TimerAlarm_r( clockid_t id,
                  const struct _itimerval * itime,
                  struct _itimerval * otime );
```

Arguments:

id

The timer type to use to implement the alarm; one of:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

itime

NULL, or a pointer to a **_itimerval** structure that specifies the length of time to wait.

otime

NULL, or a pointer to a **_itimerval** structure where the function can store the old timer trigger time.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls set an alarm signal (SIGALRM) to be delivered to the thread waiting on the timer at the time specified by *itime*. If *otime* isn't NULL, the old timer trigger time is returned.

The *TimerAlarm()* and *TimerAlarm_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3449) section for details.



- Instead of using these kernel calls directly, consider calling [alarm\(\)](#) (p. 144) or [setitimer\(\)](#) (p. 2872).
- Because of the nature of time measurement, the signal might actually get sent later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Alarm requests aren't stacked; only a single SIGALRM may be outstanding on a timer at one time. If you call *TimerAlarm()* while an alarm is outstanding, the alarm is reset to the new value passed in *itime*.

If *itime* is NULL, any previous alarm request is canceled, and no new alarm is set.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

TimerAlarm()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

TimerAlarm_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EAGAIN

All kernel timer entries are in use.

EINVAL

Invalid timer value *id*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The [alarm\(\)](#) (p. 144), [TimerAlarm\(\)](#), and [ualarm\(\)](#) (p. 3550) requests aren't stacked; only a single SIGALRM generator can be scheduled with these functions. If the SIGALRM signal hasn't been generated, the next call to [alarm\(\)](#), [TimerAlarm\(\)](#), or [ualarm\(\)](#) reschedules it.

TimerCreate(), TimerCreate_r()

Create a timer for a process

Synopsis:

```
#include <sys/neutrino.h>

int TimerCreate( clockid_t id,
                const struct sigevent *event );

int TimerCreate_r( clockid_t id,
                  const struct sigevent *event );
```

Arguments:

id

The timing base; supported types are:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

event

NULL, or a pointer to a [sigevent](#) (p. 2961) structure that contains the event to deliver when the timer fires; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *TimerCreate()* and *TimerCreate_r()* kernel calls create a per-process timer using the clock specified by *id* as the timing base.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3452) section for details.



- Instead of using these kernel calls directly, consider calling [timer_create\(\)](#) (p. 3427).
 - In order to create a timer that sends a pulse, your process must have the PROC_MGR_AID_TIMER ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
-

Use the returned timer ID in subsequent calls to the other timer functions.

The timer is created in the disabled state, and isn't enabled until you call *TimerSettime()*.

The [sigevent](#) (p. 2961) structure pointed to by *event* contains the event to deliver when the timer fires. We recommend the following event types in this case:

- If your process executes in a loop using *MsgReceivev()*, then SIGEV_PULSE is a convenient way of receiving timer pulses.
- If you use signals for event notification, note that signals are always delivered to the process and not directly to the thread that created or armed the timer. You can change this by using a *sigev_notify* of SIGEV_SIGNAL_THREAD.
- The notify types of SIGEV_UNBLOCK and SIGEV_INTR, while allowed, are of questionable use with timers. SIGEV_UNBLOCK is typically used by the *TimerTimeout()* kernel call, and SIGEV_INTR is typically used with the *InterruptWait()* kernel call.

If the *event* argument is NULL, a SIGALRM signal is sent to your process when the timer expires. To specify a handler for this signal, call [sigaction\(\)](#) (p. 2948).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

TimerCreate()

The timer ID of the newly created timer. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

TimerCreate_r()

The timer ID of the newly created timer. This function does **NOT** set *errno*.
If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EAGAIN**

All kernel timer objects are in use.

EFAULT

A fault occurred when the kernel tried to access the buffers provided.

EINVAL

The clock ID isn't valid.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

TimerDestroy(), TimerDestroy_r()

Destroy a process timer

Synopsis:

```
#include <sys/neutrino.h>

int TimerDestroy( timer_t id );
int TimerDestroy_r( timer_t id );
```

Arguments:

id

The ID of the timer that you want to destroy, as returned by [TimerCreate\(\)](#) (p. 3451).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls remove a previously created timer specified by *id*. The timer is removed from the active system timer list and returned to the list of available timers.

The *TimerDestroy()* and *TimerDestroy_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3454) section for details.



Instead of using these kernel calls directly, consider calling [timer_delete\(\)](#) (p. 3431).

If a timeout is pending when *TimerDestroy()* removes the timer, the timer is removed without being activated.

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

TimerDestroy()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

TimerDestroy_r()

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:**EINVAL**

The timer specified by *id* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

TimerInfo(), TimerInfo_r()

Get information about a timer

Synopsis:

```
#include <sys/neutrino.h>

int TimerInfo( pid_t pid,
               timer_t id,
               int flags,
               struct _timer_info* info );

int TimerInfo_r( pid_t pid,
                 timer_t id,
                 int flags,
                 struct _timer_info* info );
```

Arguments:

pid

The process ID that you're requesting the timer information for.

id

The ID of the timer, as returned by [TimerCreate\(\)](#) (p. 3451).

flags

Supported flags are:

- `_NTO_TIMER_SEARCH` — if this flag is specified and the timer ID doesn't exist, return information on the *next* timer ID. This provides a mechanism to discover all of the timers in the process.
- `_NTO_RESET_OVERRUNS` — reset the overrun count to zero in the `_timer_info` structure.
- `_NTO_TI_REPORT_TOLERANCE` — store the timer tolerance in the `otime.interval_nsec` member of the `_timer_info` structure.

info

A pointer to a `_timer_info` structure where the function can store the information about the specified timer. For more details, see “[struct _timer_info](#) (p. 3457),” below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls get information about a previously created timer specified by *id*, and store the information in the buffer pointed to by *info*.

The *TimerInfo()* and *TimerInfo_r()* functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3458) section for details.



- Instead of using these kernel calls directly, consider calling [timer_getexpstatus\(\)](#) (p. 3433), [timer_getoverrun\(\)](#) (p. 3435), or [timer_gettime\(\)](#) (p. 3437).
- In order to get timer information, your process must have the `PROCmgr_AID_TIMER` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

struct _timer_info

The `_timer_info` structure pointed to by *info* contains at least these members:

uint32_t flags

One or more of these bit flags:

_NTO_TI_ACTIVE

The timer is active.

_NTO_TI_ABSOLUTE

The timer is waiting for an absolute time to occur; otherwise, the timer is relative.

_NTO_TI_EXPIRED

The timer has expired.

_NTO_TI_PRECISE

The timer is precise, and is excluded from timer harmonization.

int32_t tid

The thread to which the timer is directed (0 if it's directed to the process).

int32_t notify

The notify type.

clockid_t clockid

The type of clock used.

uint32_t overruns

The number of overruns.

struct sigevent event

The event dispatched when the timer expires.

struct itimerspec itime

Time when the timer was started.

struct itimerspec otime

Time remaining before the timer expires. If you specified `_NTO_TI_REPORT_TOLERANCE` in the *flags* argument, *otime.interval_nsec* is set to the timer tolerance.

For more information, see the description of [TimerCreate\(\)](#) (p. 3451).

Blocking states

These calls don't block.

Returns:

The only difference between these functions is the way they indicate errors:

TimerInfo()

The ID of the timer that the information is for. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

TimerInfo_r()

The ID of the timer that the information is for. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:**EINVAL**

The timer specified by *id* doesn't exist.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

ESRCH

The process specified by *pid* doesn't exist.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

TimerSettime(), TimerSettime_r()

Set the expiration time, tolerance, or precision for a timer

Synopsis:

```
#include <sys/neutrino.h>

int TimerSettime( timer_t id,
                  int flags,
                  const struct _itimer * itime,
                  struct _itimer * oitime );

int TimerSettime_r( timer_t id,
                    int flags,
                    const struct _itimer * itime,
                    struct _itimer * oitime );
```

Arguments:

id

The ID of the timer whose expiration date you want to set, as returned by [TimerCreate\(\)](#) (p. 3451).

flags

Flags that specify what you want to set for the timer; at most one of the following:

- **TIMER_ABSTIME** — set an absolute expiration time.
- **TIMER_TOLERANCE** — specify the amount of the tolerance to allow the kernel in low-power situations.

If you don't set either of these bits, the function sets a relative expiration time. For more information, see below.

If you're setting an expiration time (i.e., you specify **TIMER_ABSTIME** or neither bit), you can OR in **TIMER_PRECISE** to exclude the timer from timer harmonization.

itime

A pointer to an `_itimer` structure that specifies the expiration date or tolerance, depending on the flag set. If you specify **TIMER_TOLERANCE** for the *flags* argument, *itime* can be `NULL`.

oitime

NULL, or a pointer to an `_itimer` structure where the function can store the previous data.

Library:

`libc`

Use the `-lc` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `TimerSettime()` and `TimerSettime_r()` kernel calls set the expiration time of the timer specified by `id`.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3462) section for details.



Instead of using these kernel calls directly, consider calling [timer_gettime\(\)](#) (p. 3437) or [timer_settime\(\)](#) (p. 3439).

The `_itimer` structure contains at least the following members:

- `uint64_t nsec`
- `uint64_t interval_nsec`

The meaning of these fields depends on the flags specified.

Expiration date

If you specify a flag of `TIMER_ABSTIME`, then the `nsec` member represents an absolute expiration date in nanoseconds from the Unix Epoch, 00:00:00 January 1, 1970 UTC. If the date specified has already passed, then the expiration event is delivered immediately.

If you don't specify `TIMER_ABSTIME` or `TIMER_TOLERANCE`, `nsec` represents a relative expiration period that's offset from the given clock's current system time, in nanoseconds.

If `nsec` is zero, then the timer is disarmed.

If `interval_nsec` is nonzero, then it specifies a repeat rate which is added to the timer once the `nsec` period has expired. Subsequently, the timer is automatically rearmed, causing it to become repetitive with a period of `interval_nsec`.

If the timer is already armed when you call `TimerSettime()`, this call discards the previous setting and sets a new setting.

If the event notification specified by `TimerCreate()` has a `sigev_code` of `SI_TIMER`, then at most one event is queued. In this case, if an event is pending from a previous

timer when the timer fires again, a timer overrun occurs. You can use the [TimerInfo\(\)](#) (p. 3456) kernel call to obtain the number of overruns that have occurred on this timer.



Because of the nature of time measurement, the timer might actually expire after the specified time. For more information, see the Tick, Tock:

Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

If the *oitime* argument isn't `NULL`, the function fills in the structure that it points to with the interval timer period (i.e., the previous amount of time left before the timer was to have expired), or zero if the timer was disarmed at the time of the call. The previous repeat rate is stored in the *interval_nsec* member.

Timer tolerance

If you specify the `TIMER_TOLERANCE` flag, then:

- If *itime* isn't `NULL`, the *nsec* member specifies the amount of tolerance, in nanoseconds, that the kernel is allowed in low-power situations. The kernel uses the expiry time plus the tolerance value to decide if it should enter tickless operation or a low-power mode.
- If *oitime* isn't `NULL`, the function sets the *nsec* member to the previous tolerance.

You can set the tolerance at any time without affecting the active/inactive status of the timer. For more information, see “Clocks, timers, and power management” in the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Blocking states

This call doesn't block.

Returns:

The only difference between these functions is the way they indicate errors:

TimerSettime()

If an error occurs, -1 is returned and [errno](#) (p. 539) is set. Any other value returned indicates success.

TimerSettime_r()

`EOK` is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

Errors:

EINVAL

The timer specified by *id* doesn't exist.

EFAULT

A fault occurred when the kernel tried to access *itime* or *oitime*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

TimerTimeout(), TimerTimeout_r()

Set a timeout on a blocking state

Synopsis:

```
#include <sys/neutrino.h>

int TimerTimeout( clockid_t id,
                  int flags,
                  const struct sigevent * notify,
                  const uint64_t * ntime,
                  uint64_t * otime );

int TimerTimeout_r( clockid_t id,
                    int flags,
                    const struct sigevent * notify,
                    const uint64_t * ntime,
                    uint64_t * otime );
```

Arguments:

id

The clock to use to implement the timeout; one of:

- **CLOCK_REALTIME** — the standard POSIX-defined clock. Timers based on this clock wake up the processor if it's in a power-saving mode.
- **CLOCK_SOFTTIME** — this clock is active only when the processor *isn't* in a power-saving mode. For example, an application using a **CLOCK_SOFTTIME** timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode.

While the processor isn't in a power-saving mode, **CLOCK_SOFTTIME** behaves the same as **CLOCK_REALTIME**.

- **CLOCK_MONOTONIC** — this clock always increases at a constant rate and can't be adjusted.

flags

Flags that specify which states the timeout applies to. For the list and description of applicable states, see the section “[Timeout states](#) (p. 3466).”

You can also OR in at most one of the following bits:

- **TIMER_ABSTIME** — set an absolute expiration time.

- **TIMER_TOLERANCE** — specify the amount of the tolerance to allow the kernel in low-power situations.

If you don't set either of these bits, the function sets a relative expiration time.

If you're setting an expiration time (i.e., you didn't specify **TIMER_TOLERANCE**), you can OR in **TIMER_PRECISE** to exclude the timer from timer harmonization.

notify

NULL, or a pointer to a [sigevent](#) (p. 2961) structure that contains the event to act on when the timeout expires. See “[Event types](#) (p. 3467)” for the list of recommended event types.

ntime

The timeout (in nanoseconds).

otime

NULL, or a pointer to a location where the function can store the time remaining in the sleep.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *TimerTimeout()* and *TimerTimeout_r()* kernel calls set a timeout on any kernel blocking state.

These functions are identical except in the way they indicate errors. See the [Returns](#) (p. 3469) section for details.



Instead of using these kernel calls directly, consider calling [timer_timeout\(\)](#) (p. 3442).

These blocking states are entered as a result of the following kernel calls:

Call	Blocking state
InterruptWait() (p. 1183)	STATE_INTR
MsgReceivev() (p. 1771)	STATE_RECEIVE

Call	Blocking state
<i>MsgSendv()</i> (p. 1801)	STATE_SEND or STATE_REPLY
<i>SignalSuspend()</i> (p. 2995)	STATE_SIGSUSPEND
<i>SignalWaitinfo()</i> (p. 2997)	STATE_SIGWAITINFO
<i>SyncCondvarWait()</i> (p. 3282)	STATE_CONDVAR
<i>SyncMutexLock()</i> (p. 3294)	STATE_MUTEX
<i>SyncSemWait()</i> (p. 3303)	STATE_SEM
<i>ThreadJoin()</i> (p. 3422)	STATE_JOIN

Timeout states

You can specify which states the timeout should apply to via a bitmask passed in the *flags* argument. The bits are defined by the following constants:

_NTO_TIMEOUT_CONDVAR

Time out on STATE_CONDVAR.

_NTO_TIMEOUT_JOIN

Time out on STATE_JOIN.

_NTO_TIMEOUT_INTR

Time out on STATE_INTR.

_NTO_TIMEOUT_MUTEX

Time out on STATE_MUTEX.

_NTO_TIMEOUT_RECEIVE

Time out on STATE_RECEIVE.

_NTO_TIMEOUT_REPLY

Time out on STATE_REPLY.

_NTO_TIMEOUT_SEM

Time out on STATE_SEM.

_NTO_TIMEOUT_SEND

Time out on STATE_SEND.

_NTO_TIMEOUT_SIGSUSPEND

Time out on STATE_SIGSUSPEND.

_NTO_TIMEOUT_SIGWAITINFO

Time out on STATE_SIGWAITINFO.

For example, to set a timeout on *MsgSendv()*, specify:

```
_NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY
```

Once a timeout is specified using *TimerTimeout()*, it's armed and released under the following conditions:

Armed

The kernel attempts to enter a blocking state specified in *flags*.

Released

One of the above kernel calls completed without blocking, *or* the kernel call blocks but unblocks before the timeout expires, *or* the timeout expires.

TimerTimeout() always operates on a one-shot basis. When one of the above kernel calls returns (or is interrupted by a signal), the timeout request is removed from the system. Only one timeout per thread may be in effect at a time. A second call to *TimerTimeout()*, without calling one of the above kernel functions, replaces the existing timeout on that thread. A call with *flags* set to zero ensures that a timeout won't occur on any state. This is the default when a thread is created.

Always call *TimerTimeout()* just before the function that you wish to time out. For example:

```
...
event.sigev_notify = SIGEV_UNBLOCK;
timeout = 10×1000000000;
TimerTimeout( CLOCK_REALTIME,
              _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY,
              &event, &timeout, NULL );
MsgSendv( coid, NULL, 0, NULL, 0 );
...
```

If the signal handler is called between the calls to *TimerTimeout()* and *MsgSendv()* (p. 1801), the *TimerTimeout()* values are saved during the signal handler and then are restored when the signal handler exits.

Event types

If the timeout expires, the kernel acts upon the event specified in the *sigevent* (p. 2961) structure pointed to by the *notify* argument. We recommend the following event types in this case:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE

- SIGEV_UNBLOCK
- SIGEV_INTR

Only SIGEV_UNBLOCK guarantees that the kernel call unblocks. A signal may be ignored, blocked, or accepted by another thread, and a pulse can only unblock a *MsgReceivev()*. If you pass NULL for *notify*, SIGEV_UNBLOCK is assumed. In this case, a timed out kernel call returns failure with an error of ETIMEDOUT.



MsgSendv() (p. 1801) doesn't unblock on SIGEV_UNBLOCK if the server has already received the message via *MsgReceivev()* (p. 1771) and has specified _NTO_CHF_UNBLOCK in the *flags* argument to its *ChannelCreate()* (p. 278) call. In this case, it's up to the server to do a *MsgReplyv()* (p. 1778).

The timeout

The type of timer used to implement the timeout is specified with the *id* argument.

The timeout:

- is specified by the *ntime* argument (the number of nanoseconds)
- is relative to the current time (when *TimerTimeout()* is called), unless *flags* includes TIMER_ABSTIME, which makes the timeout occur at the absolute time set in *ntime*
- occurs on a clock tick (see *ClockPeriod()*) (p. 329) so the actual wakeup time is a minimum of:

```
( ntime ) / ( size of timer tick ) nanoseconds
```



Because of the nature of time measurement, the timer might actually expire later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

If you don't wish to block for any time, you can pass a NULL for *ntime*, in which case no timer is used, the event is assumed to be SIGEV_UNBLOCK, and an attempt to enter a blocking state as set by *flags* immediately returns with ETIMEDOUT. Although a questionable practice, you can use it to poll potential blocking kernel calls. For example, you can poll for messages using *MsgReceivev()* with an immediate timeout. A much better approach is to use multiple threads and have one block waiting for messages.

If you set *flags* to _NTO_TIMEOUT_NANOSLEEP, then these calls block in the STATE_NANOSLEEP state until the timeout (or a signal that unblocks the thread) occurs. You can use this to implement an efficient kernel sleep as follows:

```
TimerTimeout( CLOCK_REALTIME, _NTO_TIMEOUT_NANOSLEEP,  
              NULL, ntime, otime );
```


If *otime* isn't NULL and the sleep is unblocked by a signal, it contains the time remaining in the sleep.

The actual timeout that occurred is returned in *otime*. The resolution of *otime* for both *timer_timeout()* and *TimerTimeout()* functions is in nanoseconds. The difference, however, for *otime* in these two functions is in the format. For *timer_timeout()*, the *otime* is a pointer to `timespec` structure with two integers, whereas for *TimerTimeout()*, the pointer is of `uint64_t` type.

Blocking states

These calls don't block unless you specify `_NTO_TIMEOUT_NANOSLEEP` in *flags*. In this case, the calls block as follows:

STATE_NANOSLEEP

The calling thread blocks for the requested time period.

Returns:

The only difference between these functions is the way they indicate errors:

TimerTimeout()

The previous flags. If an error occurs, -1 is returned and [errno](#) (p. 539) is set.

TimerTimeout_r()

The previous flags. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EAGAIN

All kernel timer entries are in use.

EFAULT

A fault occurred when the kernel tried to access *ntime*, *otime*, or *notify*.

EINTR

The call was interrupted by a signal.

EINVAL

Invalid timer value *id*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The timeout value *starts timing out* when *TimerTimeout()* is called, *not* when the blocking state is entered. It might be possible to get preempted after calling *TimerTimeout()* but before the blocking kernel call.

times()

Get time-accounting information

Synopsis:

```
#include <sys/times.h>

clock_t times( struct tms* buffer );
```

Arguments:

buffer

A pointer to a `tms` structure where the function can store the time-accounting information. For information about the `tms` structure, see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `times()` function stores time-accounting information in the structure pointed to by *buffer*. The type `clock_t` and the `tms` structure are defined in the `<sys/times.h>` header file.

The `tms` structure contains at least the following members:

`clock_t tms_utime`

The CPU time charged for the execution of user instructions of the calling process.

`clock_t tms_stime`

The CPU time charged for execution by the system on behalf of the calling process.

`clock_t tms_cutime`

The sum of the `tms_utime` and `tms_cutime` values of the child processes.

`clock_t tms_cstime`

The sum of the *tms_stime* and *tms_cstime* values of the child processes.

All times are in CLK_TCK'ths of a second. CLK_TCK is defined in the `<time.h>` header file. A CLK_TCK is the equivalent of:

```
#define sysconf( _SC_CLK_TCK )
```

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when a [wait\(\)](#) (p. 3652) or [waitpid\(\)](#) (p. 3666) function returns the process ID of this terminated child. If a child process hasn't waited for its terminated children, their times aren't included in its times.

Returns:

The elapsed real time, in clock ticks, of kernel uptime.



The value returned may overflow the possible range of type `clock_t`.

Examples:

```
/*
 * The following program executes the program
 * specified by argv[1]. After the child program
 * is finished, the cpu statistics of the child are
 * printed.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/times.h>

int main( int argc, char **argv )
{
    struct tms chldtim;

    system( argv[1] );
    times( &chldtim );
    printf( "system time = %d\n", chldtim.tms_cstime );
    printf( "user time   = %d\n", chldtim.tms_cutime );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

timespec

Time-specification structure

Synopsis:

```
#include <time.h>

struct timespec {
    time_t    tv_sec;
    long      tv_nsec;
}
```

Description:

The `timespec` structure specifies a time in seconds and nanoseconds. The members include:

tv_sec

The number of seconds. If specifying an absolute time, this member is the number of seconds since 1970.

tv_nsec

The number of nanoseconds.

Classification:

[*POSIX 1003.1*](#) (p. 46)

timespec2nsec()

Convert a timespec structure to nanoseconds

Synopsis:

```
#include <time.h>

uint64_t timespec2nsec( const struct timespec* ts );
```

Arguments:

ts

A pointer to the [timespec](#) (p. 3473) that you want to convert to nanoseconds.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *timespec2nsec()* function converts the number of seconds and nanoseconds in the *timespec* structure pointed to by *ts* into nanoseconds.

Returns:

The number of nanoseconds.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

timezone

The number of seconds by which the local time zone is earlier than UTC

Synopsis:

```
#include <time.h>

long int timezone;
```

Description:

This global variable holds the number of seconds by which the local time zone is earlier than Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time). Whenever you call a time function, [tzset\(\)](#) (p. 3547) is called to set the variable, based on the current time zone.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

tm

Structure that describes calendar time

Synopsis:

```
#include <time.h>
struct tm {
    int  tm_sec;
    int  tm_min;
    int  tm_hour;
    int  tm_mday;
    int  tm_mon;
    int  tm_year;
    int  tm_wday;
    int  tm_yday;
    int  tm_isdst;
    long int tm_gmtoff;
    const char * tm_zone;
};
```

Description:

The `tm` structure describes the calendar time. The members of this structure include:

tm_sec

Seconds after the minute, in the range [0,61], allowing for leap seconds.

tm_min

Minutes after the hour, in the range [0,59].

tm_hour

Hours after midnight, in the range [0,23].

tm_mday

Day of the month, in the range [1,31].

tm_mon

Months since January, in the range [0,11].

tm_year

Years since 1900.

tm_wday

Days since Sunday, in the range [0,6].

tm_yday

Days since January 1, in the range [0,365], allowing for leap years.

tm_isdst

Daylight saving time flag. This field is positive if daylight saving time is in effect, 0 if daylight saving time isn't in effect, and negative if the information isn't available.

tm_gmtoff

(QNX Neutrino extension) The offset from UTC; see [setlocale\(\)](#) (p. 2880).

tm_zone

(QNX Neutrino extension) A pointer to the time zone's name.



This pointer could become stale or invalid after the next call to [tzset\(\)](#) (p. 3547).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

tmpfile(), *tmpfile64()*

Create a temporary file

Synopsis:

```
#include <stdio.h>

FILE* tmpfile( void );
FILE* tmpfile64( void );
```

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tmpfile()* and *tmpfile64()* functions create a temporary file and open a corresponding FILE stream. The *tmpfile64()* function is a 64-bit version of *tmpfile()*. The file is automatically removed when it's closed or when the program terminates. The file is opened in update mode (as in [fopen\(\)](#) (p. 681)'s `w+` mode).

If the process is killed between file creation and unlinking, a permanent file may be left behind.



When a stream is opened in update mode, both reading and writing may be performed. However, writing may not be followed by reading without an intervening call to the [fflush\(\)](#) (p. 641) function, or to a file-positioning function ([fseek\(\)](#) (p. 772), [fsetpos\(\)](#) (p. 776), [rewind\(\)](#) (p. 2646)). Similarly, reading may not be followed by writing without an intervening call to a file-positioning function, unless the read resulted in end-of-file.

Returns:

A pointer to the stream of the temporary file, or NULL if an error occurs ([errno](#) (p. 539) is set).

Errors:

EACCES

The calling process doesn't have permission to create the temporary file.

EMFILE

The calling process already has already used OPEN_MAX file descriptors.

ENFILE

The system already has the maximum number of files open.

EROFS

The filesystem for the temporary file is read-only.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

static FILE *TempFile;

int main( void )
{
    TempFile = tmpfile();
    ...
    fclose( TempFile );

    /* The temporary file will be removed when we exit. */
    return EXIT_SUCCESS;
}
```

Classification:

tmpfile() is [ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46); *tmpfile64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

tmpnam()

Generate a unique string for use as a filename

Synopsis:

```
#include <stdio.h>

char* tmpnam( char* buffer );
```

Arguments:

buffer

NULL, or a pointer to a buffer where the function can store the filename. If *buffer* isn't NULL, the buffer must be at least *L_tmpnam* bytes long.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tmpnam()* function generates a unique string that's a valid filename and that's not the same as the name of an existing file.

The *tmpnam()* function generates up to `TMP_MAX` unique file names before it starts to recycle them:

- If a file with the new name doesn't exist, the new name is returned.
- If a file with the new name does exist, a default pathname with a basename of `000000` is returned.

The generated filename is prefixed with the first accessible directory contained in:

- The ***TMPDIR*** environment variable
- The temporary file directory *P_tmpdir* (defined in `<stdio.h>`)
- The `_PATH_TMP` constant (defined in `<paths.h>`)

If all of these paths are inaccessible, *tmpnam()* attempts to use `/tmp` and then the current working directory.

The generated filename is stored in an internal buffer; if *buffer* is NULL, the function returns a pointer to this buffer; otherwise, *tmpnam()* copies the filename into *buffer*.

Subsequent calls to *tmpnam()* reuse the internal buffer. If *buffer* is NULL, you might want to duplicate the resulting string. For example,

```
char *name1, *name2;
name1 = strdup( tmpnam( NULL ) );
name2 = strdup( tmpnam( NULL ) );
```

Returns:

A pointer to the generated filename for success, or NULL if an error occurs (*errno* (p. 539) is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char filename[L_tmpnam];
    FILE *fp;

    tmpnam( filename );
    fp = fopen( filename, "w+b" );
    ...
    fclose( fp );
    remove( filename );

    return EXIT_SUCCESS;
}
```

Classification:

ANSI (p. 46), *POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Read the <i>Caveats</i>

Caveats:

The *tmpnam()* function *isn't thread-safe* if you pass it a NULL *buffer*.

This function only creates pathnames; the application must create and remove the files.

It's possible for another thread or process to create a file with the same name between when the pathname is created and the file is opened.

tolower()

Convert a character to lowercase

Synopsis:

```
#include <ctype.h>

int tolower( int c );
```

Arguments:

c

The character that you want to convert. This must be representable as an `unsigned char` or be EOF; the behavior for other values is undefined.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *tolower()* function converts *c* to a lowercase letter, if *c* represents an uppercase letter.

Returns:

The corresponding lowercase letter when the argument is an uppercase letter; otherwise, the original character is returned. If the original character can't be represented as an `unsigned char` and isn't EOF, the behavior is undefined.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char chars[] = {
    'A',
    '5',
    '$',
    'Z'
};

#define SIZE sizeof( chars ) / sizeof( char )

int main( void )
{
    int i;
```

```
    for( i = 0; i < SIZE; i++ ) {  
        printf( "%c ", tolower( chars[ i ] ) );  
    }  
    printf( "\n" );  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
a 5 $ z
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

toupper()

Convert a character to uppercase

Synopsis:

```
#include <ctype.h>

int toupper( int c );
```

Arguments:

c

The character that you want to convert. This must be representable as an `unsigned char` or be EOF; the behavior for other values is undefined.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *toupper()* function converts *c* to a uppercase letter, if *c* represents a lowercase letter.

Returns:

The corresponding uppercase letter when the argument is a lowercase letter; otherwise, the original character is returned. If the original character can't be represented as an `unsigned char` and isn't EOF, the behavior is undefined.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char chars[] = {
    'a',
    '5',
    '$',
    'z'
};

#define SIZE sizeof( chars ) / sizeof( char )

int main( void )
{
    int i;
```



```
    for( i = 0; i < SIZE; i++ ) {  
        printf( "%c ", toupper( chars[ i ] ) );  
    }  
    printf( "\n" );  
    return EXIT_SUCCESS;  
}
```

produces the output:

```
A 5 $ Z
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

towctrans()

Convert a wide character in a specified manner

Synopsis:

```
#include <wctype.h>

wint_t towctrans( wint_t wc,
                  wctrans_t category );
```

Arguments:

- wc***
The wide character that you want to convert.
- category***
How you want to convert the character; get this by calling [wctrans\(\)](#) (p. 3720).

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *towctrans()* function converts *wc*, using the mapping described by *category*. The following functions are equivalent:

Function	Equivalent <i>wctrans()</i> call
<code>towlower(wc)</code>	<code>towctrans(wc, wctrans("tolower"))</code>
<code>towupper(wc)</code>	<code>towctrans(wc, wctrans("toupper"))</code>

Returns:

The corresponding converted wide character when the argument is valid; otherwise, the original wide character.

Errors:

EINVAL

The conversion descriptor in *category* is invalid.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

towlower()

Convert a wide character to lowercase

Synopsis:

```
#include <wctype.h>

wint_t towlower( wint_t wc );
```

Arguments:

wc

The wide character that you want to convert. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be **WEOF**; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *towlower()* function converts *wc* to a lowercase letter, if *wc* represents an uppercase letter.

Returns:

The corresponding lowercase letter when the argument is an uppercase letter; otherwise, the original wide character is returned. If the argument doesn't correspond to a valid character in the current locale and isn't **WEOF**, the function's behavior is undefined.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

towupper()

Convert a wide character to uppercase

Synopsis:

```
#include <wctype.h>

wint_t towupper( wint_t wc );
```

Arguments:

wc

The wide character that you want to convert. This argument must be a wide-character code that corresponds to a valid character in the current locale, or must be **WEOF**; otherwise, the function's behavior is undefined.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *towupper()* function converts *wc* to an uppercase letter if *wc* represents a lowercase letter.

Returns:

The corresponding uppercase letter when the argument is a lowercase letter; otherwise, the original wide character is returned. If the argument doesn't correspond to a valid character in the current locale and isn't **WEOF**, the function's behavior is undefined.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

trace_func_enter()

Insert a trace event for the entry to a function

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_func_enter( void *this_fn,
                     void *call_site );
```

Arguments:

this_fn

A pointer to the function you've entered.

call_site

The address of the calling site.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_func_enter()* function calls [TraceEvent\(\)](#) (p. 3510) to insert an event of class `_NTO_TRACE_SYSTEM` and type `_NTO_TRACE_SYS_FUNC_ENTER` into the trace log.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

trace_func_exit()

Insert a trace event for the exit from a function

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_func_exit( void *this_fn,
                    void *call_site );
```

Arguments:

this_fn

A pointer to the function you've entered.

call_site

The address of the calling site.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_func_exit()* function calls [TraceEvent\(\)](#) (p. 3510) to insert an event of class `_NTO_TRACE_SYSTEM` and type `_NTO_TRACE_SYS_FUNC_EXIT` into the trace log.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

trace_here()

Insert a trace event for the current address

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_here( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_here()* function calls [TraceEvent\(\)](#) (p. 3510) to insert an event of class `_NTO_TRACE_SYSTEM` and type `_NTO_TRACE_SYS_ADDRESS` for the current address.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Examples:**Classification:**

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

trace_logb()

Insert a user combine trace event

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_logb( int code,
                const void *buf,
                size_t nbytes );
```

Arguments:

code

The event code, which must be in the range from `_NTO_TRACE_USERFIRST` through `_NTO_TRACE_USERLAST`.

data

A pointer to the data to include with the event.

nbytes

The length of the data, in bytes. The function converts this value into the number of 32-bit integers.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `trace_()` function calls [TraceEvent\(\)](#) (p. 3510) to insert a user combine event of class `_NTO_TRACE_USER`, and of the type specified by `code`.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Examples:

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

trace_logbc()

Insert a trace event of an arbitrary class and type with arbitrary data

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_logbc( int class,
                 int event,
                 const void *data,
                 size_t nbytes );
```

Arguments:

class

The class of event that you want to insert.

event

The type of event that you want to insert.

data

A pointer to the data to include with the event.

nbytes

The length of the data, in bytes. The function converts this value into the number of 32-bit integers.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_logbc()* function calls [TraceEvent\(\)](#) (p. 3510) to insert an event of the given class and type with the given data. For information about the classes and events, see the Events and the Kernel chapter of the System Analysis Toolkit *User's Guide*.



If the length of the data isn't an even multiple of 32-bit values, or the data address isn't 32-bit aligned, the function creates a temporary copy of the data.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).**Errors:****ECANCELED**

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Classification:[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

trace_logf()

Insert a user string trace event

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_logf( int code,
                const char *fmt, ... );
```

Arguments:

code

The event code, which must be in the range from `_NTO_TRACE_USERFIRST` through `_NTO_TRACE_USERLAST`.

fmt

A [printf\(\)](#) (p. 2133)-style formatting string. This must be followed by additional arguments that provide the items required by the *fmt* string.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_logf()* function calls [TraceEvent\(\)](#) (p. 3510) to insert a user string event of class `_NTO_TRACE_USER`, and of the type specified by *code*.

Returns:

`0`

Success.

`-1`

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

It isn't safe to call this function from an interrupt handler if the formatting codes or parameters include floating-point elements.

trace_logi()

Insert a user simple trace event

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_logi( int code,
                unsigned d1,
                unsigned d2 );
```

Arguments:

code

The event code, which must be in the range from `_NTO_TRACE_USERFIRST` through `_NTO_TRACE_USERLAST`.

d1, d2

Data to include with the event.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `trace_()` function calls [TraceEvent\(\)](#) (p. 3510) to insert a user simple event of class `_NTO_TRACE_USER`, and of the type specified by `code`.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

trace_nlogf()

Insert a user string trace event, specifying a maximum string length

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_nlogf( int code,
                 int max,
                 const char *fmt, ... );
```

Arguments:

code

The event code, which must be in the range from `_NTO_TRACE_USERFIRST` through `_NTO_TRACE_USERLAST`.

max

The maximum length of the string to include in the event, in bytes, or 0 for no limit.

fmt

A [printf\(\)](#) (p. 2133)-style formatting string. This must be followed by additional arguments that provide the items required by the *fmt* string.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_nlogf()* function calls [TraceEvent\(\)](#) (p. 3510) to insert a user string event of class `_NTO_TRACE_USER`, and of the type specified by *code*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**ECANCELED**

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Examples:**Classification:**

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

It isn't safe to call this function from an interrupt handler if the formatting codes or parameters include floating-point elements.

trace_vnlogf()

Insert a user string trace event, using a variable argument list

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int trace_vnlogf( int code,
                  int max,
                  const char *fmt,
                  va_list arglist );
```

Arguments:

code

The event code, which must be in the range from `_NTO_TRACE_USERFIRST` through `_NTO_TRACE_USERLAST`.

max

The maximum length of the string to include in the event, in bytes, or 0 for no limit.

fmt

A [printf\(\)](#) (p. 2133)-style formatting string.

arglist

A variable argument list of the items required by the *fmt* string.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *trace_vnlogf()* function calls [TraceEvent\(\)](#) (p. 3510) to insert a user string event of class `_NTO_TRACE_USER`, and of the type specified by *code*.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECANCELED

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The application doesn't have permission to perform the action.

Examples:

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

It isn't safe to call this function from an interrupt handler if the formatting codes or parameters include floating-point elements.

TraceEvent()

Control the tracing of kernel events

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int TraceEvent( int cmd,
                ... );
```

Arguments:

cmd

The control action that you want to make; one of the following:

- [_NTO_TRACE_ADDALLCLASSES](#) (p. 3516)
- [_NTO_TRACE_ADDCLASS](#) (p. 3516)
- [_NTO_TRACE_ADDCLASSEVHANDLER](#) (p. 3519)
- [_NTO_TRACE_ADDEVENT](#) (p. 3516)
- [_NTO_TRACE_ADDEVENTHANDLER](#) (p. 3518)
- [_NTO_TRACE_ALLOCBUFFER](#) (p. 3513)
- [_NTO_TRACE_CLRCLASSPID](#) (p. 3516)
- [_NTO_TRACE_CLRCLASSTID](#) (p. 3516)
- [_NTO_TRACE_CLREVENTPID](#) (p. 3516)
- [_NTO_TRACE_CLREVENTTID](#) (p. 3516)
- [_NTO_TRACE_DEALLOCBUFFER](#) (p. 3513)
- [_NTO_TRACE_DELALLCLASSES](#) (p. 3516)
- [_NTO_TRACE_DELCLASS](#) (p. 3516)
- [_NTO_TRACE_DELCLASSEVHANDLER](#) (p. 3519)
- [_NTO_TRACE_DELEVENT](#) (p. 3516)
- [_NTO_TRACE_DELEVENTHANDLER](#) (p. 3518)
- [_NTO_TRACE_FLUSHBUFFER](#) (p. 3513)
- [_NTO_TRACE_INSERTCUSEREVENT](#) (p. 3520)
- [_NTO_TRACE_INSERTEVENT](#) (p. 3519)
- [_NTO_TRACE_INSERTSUSEREVENT](#) (p. 3520)
- [_NTO_TRACE_INSERTUSRSTREVENT](#) (p. 3520)
- [_NTO_TRACE_QUERYEVENTS](#) (p. 3514)
- [_NTO_TRACE_SETBUFFER](#) (p. 3514)

- [_NTO_TRACE_SETRINGMODE](#) (p. 3514)
- [_NTO_TRACE_SETLINEARMODE](#) (p. 3514)
- [_NTO_TRACE_SETALLCLASSESFAST](#) (p. 3515)
- [_NTO_TRACE_SETALLCLASSESWIDE](#) (p. 3515)
- [_NTO_TRACE_SETCLASSFAST](#) (p. 3515)
- [_NTO_TRACE_SETCLASSPID](#) (p. 3516)
- [_NTO_TRACE_SETCLASSTID](#) (p. 3516)
- [_NTO_TRACE_SETCLASSWIDE](#) (p. 3515)
- [_NTO_TRACE_SETEVENTFAST](#) (p. 3515)
- [_NTO_TRACE_SETEVENTPID](#) (p. 3516)
- [_NTO_TRACE_SETEVENTTID](#) (p. 3516)
- [_NTO_TRACE_SETEVENTWIDE](#) (p. 3515)
- [_NTO_TRACE_START](#) (p. 3514)
- [_NTO_TRACE_STOP](#) (p. 3514)
- [_NTO_TRACE_STARTNOSTATE](#) (p. 3514)

Some commands require additional arguments, as described below. In general, the arguments are grouped logically as follows:

```
TraceEvent(cmd [,class [,event]] [,p1 [,p2 [,p3 ... [,pn]]]])
```

Aside from the *cmd*, these arguments are:

class

The category of events. All the events are logically grouped into the following classes:

- [_NTO_TRACE_COMM](#) — events related to communication.
- [_NTO_TRACE_CONTROL](#) — the set of control events (e.g. time-overflow event) that are used by the communication protocol between the microkernel and `tracelogger`.
- [_NTO_TRACE_INTENTER](#), [_NTO_TRACE_INTEXIT](#) — the set of interrupt entry/exit events.
- [_NTO_TRACE_INT_HANDLER_ENTER](#), [_NTO_TRACE_INT_HANDLER_EXIT](#) — entrances to and exits from interrupt handlers.
- [_NTO_TRACE_KERCALLEENTER](#), [_NTO_TRACE_KERCALLEEXIT](#) — the set of events for entries to and exits from kernel calls.
- [_NTO_TRACE_PROCESS](#) — the set of events associated with process creation and destruction.
- [_NTO_TRACE_SYSTEM](#) — events related to the system, including those for adaptive partitioning.

- `_NTO_TRACE_THREAD`, `_NTO_TRACE_VTHREAD` — the set of events for the creation and destruction of threads (or virtual threads), and for state changes.

There are also “pseudo classes” defined as a convenience:

- `_NTO_TRACE_KERCALL` — all of the kernel-call events:
`_NTO_TRACE_KERCALLEENTER` and `_NTO_TRACE_KERCALLEEXIT`.
- `_NTO_TRACE_INT` — all of the interrupt events:
`_NTO_TRACE_INTENTER`, `_NTO_TRACE_INTEXIT`,
`_NTO_TRACE_INT_HANDLER_ENTER`, and
`_NTO_TRACE_INT_HANDLER_EXIT`.

event

The event. Because the events are grouped by *class*, the *event* must be a member of the *class* in order to be valid. For more information, see the Events and the Kernel chapter and the Current Trace Events and Data appendix of the System Analysis Toolkit *User's Guide*.

p1...pn

Any additional parameters that are required to perform the desired control action.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `TraceEvent()` function lets you control all stages of the instrumentation process, including initializing, starting, stopping, controlling filters, and inserting events. These stages are broadly grouped into the following categories:

- [Buffer and execution control](#) (p. 3513)
- [Fast/wide mode configuration](#) (p. 3514)
- [Static rules filter configuration](#) (p. 3515)
- [Dynamic rules filter configuration](#) (p. 3516)
- [User-generated trace events](#) (p. 3519)



- Filter and mode settings are made regardless of the previous settings. Use care to not accidentally override or delete previous configurations.

- In order to add handlers for trace events or allocate the instrumented kernel's trace buffers, your process must have the `PROCMGR_AID_TRACE` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Buffer and execution control

These commands control the buffer setup, and start and stop logging:

- [_NTO_TRACE_ALLOCBUFFER](#) (p. 3513)
- [_NTO_TRACE_DEALLOCBUFFER](#) (p. 3513)
- [_NTO_TRACE_FLUSHBUFFER](#) (p. 3513)
- [_NTO_TRACE_QUERYEVENTS](#) (p. 3514)
- [_NTO_TRACE_SETBUFFER](#) (p. 3514)
- [_NTO_TRACE_SETRINGMODE](#) (p. 3514)
- [_NTO_TRACE_SETLINEARMODE](#) (p. 3514)
- [_NTO_TRACE_START](#) (p. 3514)
- [_NTO_TRACE_STOP](#) (p. 3514)
- [_NTO_TRACE_STARTNOSTATE](#) (p. 3514)

`_NTO_TRACE_ALLOCBUFFER, _NTO_TRACE_DEALLOCBUFFER`

```
TraceEvent(_NTO_TRACE_ALLOCBUFFER, uint bufnum, void** linkliststart)
TraceEvent(_NTO_TRACE_DEALLOCBUFFER)
```

The allocation option creates and initializes the instrumented kernel's internal circular linked list of trace buffers. These buffers hold the emitted trace events. The extra arguments for the `_NTO_TRACE_ALLOCBUFFER` command are:

bufnum

Number of buffers to allocate.

****linkliststart***

Physical address of the beginning of the circular linked list of allocated trace buffers.

Allocated trace buffers can store 1024 simple trace events.



If your application calls `TraceEvent()` with these commands, it must must have the `PROCMGR_AID_TRACE` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The deallocation option deallocates all of the previously allocated trace buffers. All events stored in the trace buffers are lost.

`_NTO_TRACE_FLUSHBUFFER`

```
TraceEvent(_NTO_TRACE_FLUSHBUFFER)
```

Forces flushing of the buffer regardless of the trace event number it contains.

`_NTO_TRACE_QUERYEVENTS`

`TraceEvent(_NTO_TRACE_QUERYEVENTS)`

Returns the number of simple trace events that are currently stored in the trace buffer.

`_NTO_TRACE_START`, `_NTO_TRACE_STARTNOSTATE`, `_NTO_TRACE_STOP`

`TraceEvent(_NTO_TRACE_START)`
`TraceEvent(_NTO_TRACE_STARTNOSTATE)`
`TraceEvent(_NTO_TRACE_STOP)`

These commands start and stop the instrumentation process. The event stream containing the trace events is opened or closed as appropriate.

The `_NTO_TRACE_START` and `_NTO_TRACE_STARTNOSTATE` options are similar, except that the latter suppresses the initial system state information (thread IDs and names of processes).

`_NTO_TRACE_SETBUFFER`

`TraceEvent(_NTO_TRACE_SETBUFFER, paddr64_t addr, size_t size)`

Use the previously allocated physical memory, at the given address and for the given size, as the trace buffer. The intended application is to place the buffer in an area of memory that will be preserved across system resets, facilitating post-crash analysis.

`_NTO_TRACE_SETRINGMODE`

`TraceEvent(_NTO_TRACE_SETRINGMODE)`

Use the internal circular linked list in ring mode. In ring mode, the kernel stores all events in a circular fashion inside the linked list without flushing them. The maximum capturing time (without history overwriting) is determined by the number of allocated buffers, as well as by the number of generated trace events.

`_NTO_TRACE_SETLINEARMODE`

`TraceEvent(_NTO_TRACE_SETLINEARMODE)`

Use the internal circular linked list in linear mode (the default). When you use this mode, every filled-up buffer is captured and flushed immediately.

Fast/wide mode configuration

These commands control whether fast or wide events are emitted:

- [`\_NTO\_TRACE\_SETALLCLASSESFAST`](#) (p. 3515)
- [`\_NTO\_TRACE\_SETALLCLASSESWIDE`](#) (p. 3515)
- [`\_NTO\_TRACE\_SETCLASSFAST`](#) (p. 3515)
- [`\_NTO\_TRACE\_SETCLASSWIDE`](#) (p. 3515)
- [`\_NTO\_TRACE\_SETEVENTFAST`](#) (p. 3515)
- [`\_NTO\_TRACE\_SETEVENTWIDE`](#) (p. 3515)

For more information about fast and wide modes, see the System Analysis Toolkit *User's Guide*:

- For descriptions of each mode, see “Fast and wide modes” in the Events and the Kernel chapter.
- For a list of the intercepted arguments and return values for each event in both modes, see the Current Trace Events and Data appendix of the System Analysis Toolkit *User's Guide*.

_NTO_TRACE_SETALLCLASSESFAST, _NTO_TRACE_SETALLCLASSESWIDE

```
TraceEvent(_NTO_TRACE_SETALLCLASSESFAST)
TraceEvent(_NTO_TRACE_SETALLCLASSESWIDE)
```

Sets the fast/wide emitting mode for all classes and events.

_NTO_TRACE_SETCLASSFAST, _NTO_TRACE_SETCLASSWIDE

```
TraceEvent(_NTO_TRACE_SETCLASSFAST, int class)
TraceEvent(_NTO_TRACE_SETCLASSWIDE, int class)
```

Sets the fast/wide emitting mode for all events within the specified *class*.

_NTO_TRACE_SETEVENTFAST, _NTO_TRACE_SETEVENTWIDE

```
TraceEvent(_NTO_TRACE_SETEVENTFAST, int class, int event)
TraceEvent(_NTO_TRACE_SETEVENTWIDE, int class, int event)
```

Sets the fast/wide emitting mode for the specified *event* for the specified *class*.

Static rules filter configuration

These commands control the operation of the static rules filter:

- [_NTO_TRACE_ADDALLCLASSES](#) (p. 3516)
- [_NTO_TRACE_ADDCLASS](#) (p. 3516)
- [_NTO_TRACE_ADDEVENT](#) (p. 3516)
- [_NTO_TRACE_CLRCLASSPID](#) (p. 3516)
- [_NTO_TRACE_CLRCLASSTID](#) (p. 3516)
- [_NTO_TRACE_CLREVENTPID](#) (p. 3516)
- [_NTO_TRACE_CLREVENTTID](#) (p. 3516)
- [_NTO_TRACE_DELALLCLASSES](#) (p. 3516)
- [_NTO_TRACE_DELCLASS](#) (p. 3516)
- [_NTO_TRACE_DELEVENT](#) (p. 3516)
- [_NTO_TRACE_SETCLASSPID](#) (p. 3516)
- [_NTO_TRACE_SETCLASSTID](#) (p. 3516)
- [_NTO_TRACE_SETEVENTPID](#) (p. 3516)
- [_NTO_TRACE_SETEVENTTID](#) (p. 3516)

For more information about this filter, see the Filtering chapter in the System Analysis Toolkit *User's Guide*.

_NTO_TRACE_ADDALLCLASSES, _NTO_TRACE_DELALLCLASSES

```
TraceEvent(_NTO_TRACE_ADDALLCLASSES)
TraceEvent(_NTO_TRACE_DELALLCLASSES)
```

Emit/suppress trace events for all classes and events.

_NTO_TRACE_ADDCLASS, _NTO_TRACE_DELCLASS

```
TraceEvent(_NTO_TRACE_ADDCLASS, class)
TraceEvent(_NTO_TRACE_DELCLASS, class)
```

Emit/suppress all trace events from a specific *class*.

_NTO_TRACE_ADDEVENT, _NTO_TRACE_DELEVENT

```
TraceEvent(_NTO_TRACE_ADDEVENT, class, event)
TraceEvent(_NTO_TRACE_DELEVENT, class, event)
```

Emit/suppress a trace *event* from a specific *class*.

**_NTO_TRACE_SETCLASSPID, _NTO_TRACE_CLRCLASSPID,
_NTO_TRACE_SETCLASSTID, _NTO_TRACE_CLRCLASSTID**

```
TraceEvent(_NTO_TRACE_SETCLASSPID, int class, pid_t pid)
TraceEvent(_NTO_TRACE_CLRCLASSPID, int class)
TraceEvent(_NTO_TRACE_SETCLASSTID, int class, pid_t pid, tid_t tid)
TraceEvent(_NTO_TRACE_CLRCLASSTID, int class)
```

Emit/suppress all events from a specified process ID (and thread ID).

**_NTO_TRACE_SETEVENTPID, _NTO_TRACE_CLREVENTPID,
_NTO_TRACE_SETEVENTTID, _NTO_TRACE_CLREVENTTID**

```
TraceEvent(_NTO_TRACE_SETEVENTPID, int class, int event, pid_t pid)
TraceEvent(_NTO_TRACE_CLREVENTPID, int class, int event)
TraceEvent(_NTO_TRACE_SETEVENTTID, int class, int event,
           pid_t pid, tid_t tid)
TraceEvent(_NTO_TRACE_CLREVENTTID, int class, int event)
```

Emit/suppress a specific *event* for a specified process ID (and thread ID).

Dynamic rules filter configuration

These commands control the operation of the dynamic rules filter, in which you set up handlers for the events you're interested in:

- [_NTO_TRACE_ADDCLASSEVHANDLER](#) (p. 3519)
- [_NTO_TRACE_ADDEVENTHANDLER](#) (p. 3518)
- [_NTO_TRACE_DELCLASSEVHANDLER](#) (p. 3519)
- [_NTO_TRACE_DELEVENTHANDLER](#) (p. 3518)

For more information about this filter, see the Filtering chapter in the System Analysis Toolkit *User's Guide*.

Accessing data in the event handler

You can access the trace event information from within the event handler by using members of the `event_data_t` structure.

The layout of the `event_data_t` structure (declared in `<sys/trace.h>`) is as follows:

```
/* event data filled by an event handler */
typedef struct
{
    __traceentry header; /* same as traceevent header */
    uint32_t* data_array; /* initialized by the user */
    uint32_t el_num; /* number of elements returned */
    void* area; /* user data */
    uint32_t feature_mask; /* bits indicate valid features */
    uint32_t feature[_NTO_TRACE_FI_NUM]; /* feature array
                                         - additional data */
} event_data_t;
```

The `event_data_t` structure includes a pointer to an array for the data arguments of the event. You *must* provide an array, and it must be large enough to hold the data for the event or events that you're handling (see the Current Trace Events and Data appendix of the System Analysis Toolkit *User's Guide*). For example:



```
event_data_t e_d_1;
uint32_t data_array_1[20]; /* 20 elements for potential args. */
e_d_1.data_array = data_array_1;
```

If you don't provide the data array, or it isn't big enough, your data segment could become corrupted.

The bits of the member `feature_mask` are related to any additional features (arguments) that could be accessed inside the event handler. All standard data arguments — the ones that correspond to the data arguments of the trace event — are delivered without changes within the `data_array`.

There are two constants associated with each additional feature:

- `_NTO_TRACE_FM***` — feature parameter masks
- `_NTO_TRACE_FI***` — feature index parameters

The currently defined features are:

Feature	Parameter mask	Index
Process ID	<code>_NTO_TRACE_FMPID</code>	<code>_NTO_TRACE_FIPID</code>
Thread ID	<code>_NTO_TRACE_FMTID</code>	<code>_NTO_TRACE_FITID</code>

If any particular bit of the `feature_mask` is set to 1, then you can access the feature corresponding to this bit within the `feature` array. Otherwise, you must not access the feature.

Macros for use with the event header

You can use the following macros, defined in `<sys/trace.h>`, to work with the header of an event:

_NTO_TRACE_GETEVENT_C(c)

Get the class.

_NTO_TRACE_GETEVENT(c)

Get the type of event.

_NTO_TRACE_GETCPU(h)

Get the number of the CPU that the event occurred on.

_NTO_TRACE_SETEVENT_C(c,cl)

Set the class in the header *c* to be *cl*.

_NTO_TRACE_SETEVENT(c, e)

Set the event type in the header *c* to be *e*.

_NTO_TRACE_ADDEVENTHANDLER, _NTO_TRACE_DELEVENTHANDLER

```
TraceEvent(_NTO_TRACE_ADDEVENTHANDLER,
            class,
            event,
            int (*event_hdlr)(event_data_t*),
            event_data_t* data_struct)
TraceEvent(_NTO_TRACE_DELEVENTHANDLER,
            class,
            event)
```

Attaches/deletes the event handler for a specified class and event, where:

event_hdlr

Pointer to the event handler.

data_struct

Pointer to the data structure *event_data_t*.

Before attaching an event handler, the thread must:

- have the `PROCMGR_AID_TRACE` and `PROCMGR_AID_IO` abilities enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do all of the above, the attachment fails with an error code of `EPERM`.

In order to emit an event data, a dynamic filter (event handler) has to return a non-zero value. If both types of the dynamic filters (event handler and class event handler) are

applicable to a particular event, the event is emitted if both event handlers return non-zero values.

`_NTO_TRACE_ADDCLASSEVHANDLER, _NTO_TRACE_DELCLASSEVHANDLER`

```
TraceEvent(_NTO_TRACE_ADDCLASSEVHANDLER,
           class,
           int (*event_hdlr)(event_data_t*),
           event_data_t* data_struct)

TraceEvent(_NTO_TRACE_DELCLASSEVHANDLER,
           class)
```

Attaches/deletes the event handler for a specified class, where:

event_hdlr

Pointer to the event handler.

data_struct

Pointer to the data structure `event_data_t`.

Before attaching an event handler, the thread must:

- have the `PROCMGR_AID_TRACE` and `PROCMGR_AID_IO` abilities enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).
- request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do all of the above, the attachment fails with an error code of `EPERM`.

In order to emit an event data, a dynamic filter (event handler) has to return a non-zero value. If both types of the dynamic filters (event handler and class event handler) are applicable to a particular event, the event is emitted if both event handlers return non-zero values.

User-generated trace events

These commands let you insert your own events into the event stream:

- [_NTO_TRACE_INSERTCUSEREVENT](#) (p. 3520)
- [_NTO_TRACE_INSERTEVENT](#) (p. 3519)
- [_NTO_TRACE_INSERTSUSEREVENT](#) (p. 3520)
- [_NTO_TRACE_INSERTUSRSTREVENT](#) (p. 3520)

`_NTO_TRACE_INSERTEVENT`

```
TraceEvent(_NTO_TRACE_INSERTEVENT, int head, int stamp,
           int data0, int data1)
```

Inserts an event of an arbitrary type and class into the event stream. It's powerful, but because the API doesn't do any of the interpretation for you, you should use this

function (with care!) only if you're an advanced user. You'll have to modify the data-interpretation program to properly interpret the event.

The arguments are:

head

Header of the trace event.

stamp

Time stamp.

data0

Data d0.

data1

Data d1.

Instead of using the `_NTO_TRACE_INSERTEVENT` command directly, you can use the following convenience functions:

***trace_func_enter()* (p. 3492)**

Insert a trace event for the entry to a function

***trace_func_exit()* (p. 3494)**

Insert a trace event for the exit from a function

***trace_here()* (p. 3496)**

Insert a trace event for the current address

***trace_loghc()* (p. 3500)**

Insert a trace event of an arbitrary class and type with arbitrary data

**`_NTO_TRACE_INSERTSUSEREVENT, _NTO_TRACE_INSERTCUSEREVENT,
_NTO_TRACE_INSERTUSRSTREVENT`**

`TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, int event, int data0,
int data1)`

`TraceEvent(_NTO_TRACE_INSERTCUSEREVENT, int event, unsigned * buf,
unsigned len)`

`TraceEvent(_NTO_TRACE_INSERTUSRSTREVENT, int event, const char * str)`

These commands insert user-created events into the event stream. Because the API handles details such as timestamping, they're reasonably easy to use. The class is `_NTO_TRACE_USER`.

_NTO_TRACE_INSERTUSEREVENT

Add a user simple event containing the data defined by *data0* and *data1*.

_NTO_TRACE_INSERTCUSEREVENT

Add a user combine event containing the data that *buf* holds.



The *len* argument represents the number of *integers* (not bytes) in *buf*.

_NTO_TRACE_INSERTUSRSTREVENT

Add a user string event containing the null-terminated string defined by *str*.

The *event* argument must be in the range from `_NTO_TRACE_USERFIRST` through `_NTO_TRACE_USERLAST`, but you can decide what each event means.

Instead of using these commands directly, you can use the following convenience functions:

***trace_logb()* (p. 3498)**

Insert a user combine trace event

***trace_logf()* (p. 3502)**

Insert a user string trace event

***trace_logi()* (p. 3504)**

Insert a user simple trace event

***trace_nlogf()* (p. 3506)**

Insert a user string trace event, specifying a maximum string length

***trace_vnlogf()* (p. 3508)**

Insert a user string trace event, using a variable argument list

Returns:**If *cmd* is set to `_NTO_TRACE_QUERYEVENTS`**

Number of events in the buffer, or -1 if an error occurs (*errno* (p. 539) is set).

If *cmd* isn't set to `_NTO_TRACE_QUERYEVENTS`

0 for success, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:**ECANCELED**

The requested action has been canceled.

EFAULT

Bad internal trace buffer address. The requested action has been specified out of order.

ENOMEM

Insufficient memory to allocate the trace buffers.

ENOTSUP

The requested action isn't supported.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Examples:

See the Tutorials chapter of the System Analysis Toolkit *User's Guide*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

You can call *TraceEvent()* from an interrupt/event handler. However, not all commands are valid in this case. The valid commands are:

- `_NTO_TRACE_INSERTSUSEREVENT`
- `_NTO_TRACE_INSERTCUSEREVENT`
- `_NTO_TRACE_INSERTUSRSTREVENT`
- `_NTO_TRACE_INSERTEVENT`
- `_NTO_TRACE_STOP`

- `_NTO_TRACE_STARTNOSTATE`
- `_NTO_TRACE_START`

traceparser()

Process trace data

Synopsis:

```
#include <sys/traceparser.h>

int traceparser (
    struct traceparser_state * stateptr,
    void * userdata,
    const char * filename );
```

Arguments:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

userdata

A pointer to arbitrary user data to pass to any event-processing callback that doesn't have its own user data.

filename

The name of the trace file that you want to parse. You can create this file by using `tracelogger`, [TraceEvent\(\)](#) (p. 3510), or some combination of the two.

Library:

libtraceparser

Use the `-l traceparser` option to `qcc` to link against this library.

Description:

The *traceparser()* function starts the parsing of the trace data in *filename*. You'll use this function if you're creating your own utility for parsing trace data (as an alternative to `traceprinter`).

Before calling this function, you must have called:

- [traceparser_init\(\)](#) (p. 3537) to initialize the parser
- [traceparser_cs\(\)](#) (p. 3526), [traceparser_cs_range\(\)](#) (p. 3529), or both to set up callbacks
- [traceparser_debug\(\)](#) (p. 3531) to set the debugging mode (optional)

When you've finished parsing the data, call [traceparser_destroy\(\)](#) (p. 3533) to destroy the parser.

Returns:

0

Success.

-1

Failure; [errno](#) (p. 539) is set. See also [traceparser_get_info\(\)](#) (p. 3534) for further details.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

traceparser_cs()

Attach a callback to a trace event

Synopsis:

```
#include <sys/traceparser.h>

int traceparser_cs (
    struct traceparser_state * stateptr,
    void * userdata,
    tracep_callb_func_t funcptr,
    unsigned class,
    unsigned event );
```

Arguments:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

userdata

A pointer to arbitrary user data that you want to pass to the callback function.

funcptr

A pointer to the callback function that you want to register. For more information, see “[Callback function](#) (p. 3527),” below.

class

The class of trace events that you want to invoke the callback function for.

event

The specific trace event that you want to invoke the callback function for. This event must be a member of the given class.

Library:

libtraceparser

Use the -l traceparser option to qcc to link against this library.

Description:

The *traceparser_cs()* function attaches the callback function specified by the pointer *funcptr*, to one particular *event*, from one particular *class*. You'll use this function if

you're creating your own utility for parsing trace data (as an alternative to `traceprinter`).

The user data (*userdata*) is passed to the callback function upon execution. The *stateptr* is an opaque structure obtained from [traceparser_init\(\)](#) (p. 3537).

Callback function

The synopsis of the callback function is defined by the `tracep_callback_func_t` data type as follows:

```
int my_callback( struct traceparser_state* statepointer,
                 void * userdata,
                 unsigned header,
                 unsigned timestamp,
                 unsigned* buffer,
                 unsigned buffer_len)
```

The arguments are:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

userdata

A pointer to arbitrary user data that you want to pass to the callback function.

header

The header of the event. For information about some macros for extracting information from this member, see “[Macros for use with the event header](#) (p. 3517)” in the entry for *TraceEvent()*.

timestamp

The time of the event.

buffer

A pointer to a buffer for the callback to use.

buffer_len

The length of the buffer, in bytes.

The callback should return 0 to indicate success, and -1 to indicate failure.

Returns:

0

Success.

-1

Failure; [errno](#) (p. 539) is set. See also [traceparser_get_info\(\)](#) (p. 3534) for further details.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

traceparser_cs_range()

Attach a callback to a range of events

Synopsis:

```
#include <sys/traceparser.h>

int traceparser_cs_range (
    struct traceparser_state * stateptr,
    void * userdata,
    tracep_callb_func_t funcptr,
    unsigned class,
    unsigned firstevent,
    unsigned lastevent );
```

Arguments:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

userdata

A pointer to arbitrary user data that you want to pass to the callback function.

funcptr

A pointer to the callback function that you want to register. For more information, see “[Callback function](#) (p. 3527)” in the entry for *traceparser_cs()*.

class

The class of trace events that you want to invoke the callback function for.

firstevent, lastevent

The range of trace events that you want to invoke the callback function for. The events must be members of the given class.

Library:

libtraceparser

Use the -l traceparser option to qcc to link against this library.

Description:

The *traceparser_cs_range()* function attaches a callback function, given by the pointer *funcptr*, to a range of events from *firstevent* through to *lastevent* inclusive, from one particular *class*. You'll use this function if you're creating your own utility for parsing trace data (as an alternative to `traceprinter`).

The user data (*userdata*) is passed to the registered callback function (*funcptr*) upon execution. The *stateptr* is an opaque structure obtained from [traceparser_init\(\)](#) (p. 3537).

Returns:

0

Success.

-1

Failure; *errno* (p. 539) is set. See also [traceparser_get_info\(\)](#) (p. 3534) for further details

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

traceparser_debug()

Set the traceparser debug modes

Synopsis:

```
#include <sys/traceparser.h>

int traceparser_debug (
    struct traceparser_state * stateptr,
    FILE * streamptr,
    unsigned flags );
```

Arguments:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

streamptr

A pointer to the debug output stream.

flags

Flags that indicate the types of debugging information to emit; one of:

- `_TRACEPARSER_DEBUG_ALL` — everything
- `_TRACEPARSER_DEBUG_NONE` — nothing (the default)

or a combination of the following:

- `_TRACEPARSER_DEBUG_ERRORS` — critical errors
- `_TRACEPARSER_DEBUG_EVENTS` — raw input events
- `_TRACEPARSER_DEBUG_HEADER` — header information
- `_TRACEPARSER_DEBUG_SYSPAGE` — syspage data

Library:

`libtraceparser`

Use the `-l traceparser` option to `qcc` to link against this library.

Description:

The *traceparser_debug()* function sets the debugging mode of the traceparser module. You can use this function if you're creating your own utility for parsing trace data (as an alternative to *traceprinter*).

Returns:

0

Success.

-1

Failure; *errno* (p. 539) is set. See also *traceparser_get_info()* (p. 3534) for further details.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

traceparser_destroy()

Destroy a traceparser state structure

Synopsis:

```
#include <sys/traceparser.h>

void traceparser_destroy (
    struct traceparser_state ** stateptr );
```

Arguments:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

Library:

libtraceparser

Use the -l traceparser option to qcc to link against this library.

Description:

The *traceparser_destroy()* function destroys the previously initialized traceparser state structure that *stateptr* points to. You'll use this function if you're creating your own utility for parsing trace data (as an alternative to *traceprinter*).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

traceparser_get_info()

Get information about traceparser

Synopsis:

```
#include <sys/traceparser.h>

void * traceparser_get_info (
    struct traceparser_state * stateptr,
    info_modes_t infomodes,
    unsigned * len );
```

Arguments:

stateptr

A pointer to the parser's state information, obtained by calling [traceparser_init\(\)](#) (p. 3537).

infomodes

A `_TRACEPARSER_INFO_*` value (defined in `traceparser.h`) that indicates what information you want to get; see below.

len

NULL, or a pointer to a location where the function can store the length of the data returned.

Library:

`libtraceparser`

Use the `-l traceparser` option to `qcc` to link against this library.

Description:

The `traceparser_get_info()` function gets information related to the state of the traceparser. You can use this function if you're creating your own utility for parsing trace data (as an alternative to `traceprinter`).

The `len` argument is a pointer to the size of the return buffer. When specified, its contents are changed to indicate the size of the return. This is primarily for the `_TRACEPARSER_SYSPAGE` and `_TRACEPARSER_HEADER_KEYWORDS` modes but it'll work for all the modes. For most of the modes, `len` may be NULL.

The `stateptr` is an opaque structure obtained from [traceparser_init\(\)](#) (p. 3537).

User info modes for *info_modes*

The following are valid user info modes; see the list below for others.

Value for <i>info_modes</i> and Pointer to return data type, cast as void *	Description
<code>_TRACEPARSER__INFO_SYSPAGE,</code> <code>syspage_entry</code>	Returns a pointer to the syspage entry.
<code>_TRACEPARSER_INFO_ENDIAN_CONV,</code> <code>unsigned</code>	Returns a dereferenced pointer; 1 if the endian conversion has been applied, 0 if no conversion has been performed.
<code>_TRACEPARSER_INFO_NOW_CALLBACK_CLASS,</code> <code>_TRACEPARSER_INFO_NOW_CALLBACK_EVENT,</code> <code>unsigned</code>	Returns the class or event numerical value of the currently executed callback function. The numerical values are considered opaque and should be used only for other traceparser functions.
<code>_TRACEPARSER_INFO_PREV_CALLBACK_CLASS,</code> <code>_TRACEPARSER_INFO_PREV_CALLBACK_EVENT,</code> <code>unsigned</code>	As above, but returns the class or event numerical value of the previously executed callback function.
<code>_TRACEPARSER_INFO_PREV_CALLBACK_RETURN,</code> <code>int</code>	Returns the value of the previously executed callback function.
<code>_TRACEPARSER_INFO_DEBUG,</code> <code>unsigned</code>	Returns the debugging flags; see traceparser_debug() (p. 3531).
<code>_TRACEPARSER_INFO_ERROR,</code> <code>unsigned</code>	Returns the traceparser error level, which you can use to diagnose errors related to the traceparser library. See <code><sys/traceparser.h></code> for a list of the returned error values.

Other valid user info modes

The following modes return a pointer (cast as void *) to the appropriate information from the traceparser:

- `_TRACEPARSER_INFO_FILE_NAME`
- `_TRACEPARSER_INFO_DATE`
- `_TRACEPARSER_INFO_VER_MAJOR`
- `_TRACEPARSER_INFO_VER_MINOR`
- `_TRACEPARSER_INFO_LITTLE_ENDIAN`
- `_TRACEPARSER_INFO_BIG_ENDIAN`
- `_TRACEPARSER_INFO_MIDDLE_ENDIAN`

- `_TRACEPARSER_INFO_ENCODING`
- `_TRACEPARSER_INFO_BOOT_DATE`
- `_TRACEPARSER_INFO_CYCLES_PER_SEC`
- `_TRACEPARSER_INFO_CPU_NUM`
- `_TRACEPARSER_INFO_SYSNAME`
- `_TRACEPARSER_INFO_NODENAME`
- `_TRACEPARSER_INFO_SYS_RELEASE`
- `_TRACEPARSER_INFO_SYS_VERSION`
- `_TRACEPARSER_INFO_MACHINE`
- `_TRACEPARSER_INFO_SYSPAGE_LEN`

Returns:

A pointer (cast as `void *`) to the requested information, or `NULL` if an error occurred ([errno](#) (p. 539) is set; you can use the `_TRACEPARSER_INFO_ERROR` mode of this function to get more details).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

traceparser_init()

Initialize the traceparser state information

Synopsis:

```
#include <sys/traceparser.h>

struct traceparser_state * traceparser_init (
    struct traceparser_state * stateptr );
```

Arguments:

stateptr

NULL or a pointer to a state-information structure obtained from a previous call to *traceparser_init()*.

Library:

libtraceparser

Use the -l traceparser option to gcc to link against this library.

Description:

The *traceparser_init()* function initializes the state of the traceparser library and returns a pointer to an initialized state structure. You'll use this function if you're creating your own utility for parsing trace data (as an alternative to *traceprinter*).

The *traceparser_state* structure is an opaque structure for use by the other traceparser functions.

Returns:

A pointer to a valid initialized state structure, or NULL if an error occurred (*errno* (p. 539) is set; see also *traceparser_get_info()* (p. 3534) for further details).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	No

truncate()*, *truncate64()

Truncate a file to a specified length

Synopsis:

```
#include <unistd.h>

int truncate( const char* path,
              off_t length );

int truncate64( const char* path,
                off64_t length );
```

Arguments:

path

The path name of the file that you want to truncate.

length

The new size of the file.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *truncate()* and *truncate64()* functions cause the regular file named by *path* to have a size of *length* bytes. The *truncate64()* function is a large-file version of *truncate()*.

The effect of *truncate()* on other types of files is unspecified. If the file previously was larger than *length*, the extra data is lost. If it was previously shorter than *length*, bytes between the old and new lengths are read as zeroes. The process must have write permission for the file.

If the request would cause the file size to exceed the soft file size limit for the process, the request fails and the implementation generates the SIGXFSZ signal for the process.

This function doesn't modify the file offset for any open file descriptions associated with the file. On successful completion, if the file size is changed, *truncate()* marks for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the *S_ISUID* and *S_ISGID* bits of the file mode may be cleared.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set.

Errors:**EACCES**

A component of the path prefix denies search permission, or write permission is denied on the file.

EFAULT

The *path* argument points outside the process's allocated address space.

EFBIG

The *length* argument was greater than the maximum file size.

EINTR

A signal was caught during execution.

EINVAL

The *length* argument is invalid, or the *path* argument isn't an ordinary file.

EIO

An I/O error occurred while reading from or writing to a filesystem.

EISDIR

The named file is a directory.

ELOOP

Too many symbolic links were encountered in resolving *path*.

EMFILE

The maximum number of file descriptors available to the process has been reached.

EMULTIHOP

Components of *path* require hopping to multiple remote machines and filesystem type doesn't allow it.

ENAMETOOLONG

The length of the specified pathname exceeds PATH_MAX bytes, or the length of a component of the pathname exceeds NAME_MAX bytes.

ENFILE

Additional space couldn't be allocated for the system file table.

ENOENT

A component of *path* doesn't name an existing file or *path* is an empty string.

ENOLINK

The *path* argument points to a remote machine and the link to that machine is no longer active.

ENOTDIR

A component of the path prefix of *path* isn't a directory.

EROFS

The named file resides on a read-only filesystem.

Classification:

truncate() is [POSIX 1003.1 XSI](#) (p. 46); *truncate64()* is [Large-file support](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ttyname()

Get a fully qualified pathname for a file

Synopsis:

```
#include <unistd.h>

char *ttyname( int fildes );
```

Arguments:

fildes

A file descriptor that's associated with the file whose name you want to get.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ttyname()* function returns a pointer to a static buffer that contains a fully qualified pathname associated with the file associated with *fildes*.

Returns:

A pointer to the pathname for *fildes*, or NULL if an error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

The device driver was in use by another process, or the driver was unable to carry out the request due to an outstanding command in progress.

EBADF

The *fildes* argument isn't a valid file descriptor.

EINTR

The function was interrupted by a signal.

ENOSYS

The *ttyname()* function isn't implemented for the filesystem specified by *filesdes*.

ENOTTY

The *filesdes* argument doesn't refer to a tty.

EPERM

The process doesn't have sufficient permission to carry out the requested command.

Examples:

```
/*
 * The following program prints out the name
 * of the terminal associated with stdin.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    if( isatty( 0 ) ) {
        printf( "%s\n", ttyname( 0 ) );
    } else {
        printf( "\n" );
    }
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

ttyname_r()

Get a fully qualified pathname for a file

Synopsis:

```
#include <unistd.h>

int ttyname_r( int fildes,
               char* name,
               size_t namesize );
```

Arguments:

fildes

A file descriptor that's associated with the file whose name you want to get.

name

A pointer to a buffer where the function can store the path name.

namesize

The size of the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ttyname_r()* function stores the null-terminated pathname of the terminal associated with the file descriptor *fildes* in the character array referenced by *name*. The array is *namesize* characters long and should have space for the name and the terminating NULL character.

Returns:

Zero for success, or an error number.

Errors:

EAGAIN

The device driver was in use by another process, or the driver was unable to carry out the request due to an outstanding command in progress.

EBADF

The *filides* argument isn't a valid file descriptor.

EINTR

The function was interrupted by a signal.

ENOSYS

The *ttyname_r()* function isn't implemented for the filesystem specified by *filedes*.

ENOTTY

The *filides* argument doesn't refer to a tty.

EPERM

The process doesn't have sufficient permission to carry out the requested command.

ERANGE

The value of *namesize* is smaller than the length of the string to be returned, including the terminating null character.

Classification:

[POSIX 1003.1 TSF](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

tzname

The abbreviations for the time zone for standard and daylight savings time

Synopsis:

```
#include <time.h>

char *tzname[];
```

Description:

This global variable holds the standard abbreviations for the time zone and the time zone when daylight saving time is in effect. Whenever you call a time function, [tzset\(\)](#) (p. 3547) is called to set the values in the array, based on the current time zone.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

tzset()

Set the time according to the current time zone

Synopsis:

```
#include <time.h>

void tzset( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *tzset()* function sets the global variables *daylight*, *timezone* and *tzname* according to the value of the **TZ** environment variable, or to the value of the `_CS_TIMEZONE` configuration string if **TZ** isn't set, or to UTC0 if neither is set.

The global variables have the following values after *tzset()* is executed:

daylight (p. 396)

Zero indicates that daylight saving time isn't supported in the locale; a nonzero value indicates that daylight saving time is supported in the locale. This variable is cleared or set after a call to the *tzset()* function, depending on whether or not a daylight saving time abbreviation is specified in the **TZ** environment variable.

timezone (p. 3475)

The number of seconds that the local time zone is earlier than Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

tzname (p. 3546)

A two-element array pointing to strings giving the abbreviations for the name of the time zone when standard and daylight saving time are in effect.

The time that you set on the computer with the `date` command reflects Coordinated Universal Time (UTC). The environment variable **TZ** is used to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the QNX Neutrino *User's Guide*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void print_zone()
{
    char *tz;

    printf( "TZ: %s\n", (tz = getenv( "TZ" ))
        ? tz : "default EST5EDT" );
    printf( " daylight: %d\n", daylight );
    printf( " timezone: %ld\n", timezone );
    printf( " time zone names: %s %s\n",
        tzname[0], tzname[1] );
}

int main( void )
{
    tzset();
    print_zone();
    setenv( "TZ", "PST8PDT", 1 );
    tzset();
    print_zone();
    return EXIT_SUCCESS;
}
```

produces the output:

```
TZ: default EST5EDT
 daylight: 1
 timezone: 18000
 time zone names: EST EDT
TZ: PST8PDT
 daylight: 1
 timezone: 28800
 time zone names: PST PDT
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 23

U

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “U”.

ualarm()

Schedule an alarm

Synopsis:

```
#include <unistd.h>

useconds_t ualarm( useconds_t usec,
                  useconds_t interval );
```

Arguments:

usec

The number of microseconds that you want to elapse before the first alarm occurs, or 0 to cancel any previous request for an alarm.

interval

The number of microseconds that you want to elapse before the subsequent alarms occur.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ualarm()* function causes the system to send the calling process a `SIGALRM` signal after *usec* microseconds of real-time have elapsed. The alarm is then sent every *interval* microseconds after that.

Processor scheduling delays may cause a delay between when the signal is sent and when the process actually handles it.

If *usec* is 0, any previous *ualarm()* request is canceled.



Because of the nature of time measurement, the signal might actually get sent later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

Returns:

0

There was no previous *ualarm()* request.

-1

An error occurred (*errno* (p. 539) is set).

Any other value

The number of microseconds until the next scheduled SIGALRM.

Errors:**EAGAIN**

All timers are in use; wait for a process to release one and try again.

Examples:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    useconds_t timeleft;

    printf( "Set the alarm and sleep\n" );
    ualarm( (useconds_t)( 10 * 1000 * 1000 ), 0 );
    sleep( 5 ); /* go to sleep for 5 seconds */

    /*
     * To get the time left before the SIGALRM is
     * to arrive, one must cancel the initial timer,
     * which returns the amount of time it had
     * remaining.
     */
    timeleft = ualarm( 0, 0 );
    printf( "Time left before cancel, and rearm: %ld\n",
           timeleft );

    /*
     * Start a new timer that kicks us when timeleft
     * seconds have passed.
     */
    ualarm( timeleft, 0 );

    /*
     * wait until we receive the SIGALRM signal; any
     * signal kills us, though, since we don't have
     * a signal handler.
     */
    printf( "Hanging around, waiting to exit\n" );
    pause();

    /* You'll never get here. */
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

[alarm\(\)](#) (p. 144), [TimerAlarm\(\)](#) (p. 3448), and [ualarm\(\)](#) requests aren't “stacked”; only a single SIGALRM generator can be scheduled with these functions. If the SIGALRM signal hasn't been generated, the next call to [alarm\(\)](#), [TimerAlarm\(\)](#), or [ualarm\(\)](#) reschedules it.

Don't mix calls to [ualarm\(\)](#) with [nanosleep\(\)](#) (p. 1855), [sleep\(\)](#) (p. 3028), [timer_create\(\)](#) (p. 3427), [timer_delete\(\)](#) (p. 3431), [timer_getoverrun\(\)](#) (p. 3435), [timer_gettime\(\)](#) (p. 3437), [timer_settime\(\)](#) (p. 3439), or [usleep\(\)](#) (p. 3586).

UDP

Internet User Datagram Protocol

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_DGRAM,
            0 );
```

Description:

UDP is a simple, unreliable datagram protocol that's used to support the `SOCK_DGRAM` abstraction for the Internet protocol family. UDP sockets are connectionless and are normally used with the [sendto\(\)](#) (p. 2837) and [recvfrom\(\)](#) (p. 2536) calls, although you can also use the [connect\(\)](#) (p. 348) call to fix the destination for future packets (in which case you can use the [recv\(\)](#) (p. 2533) or [read\(\)](#) (p. 2502) and [send\(\)](#) (p. 2829) or [write\(\)](#) (p. 3737) system calls).

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space; that is, a UDP port may *not* be “connected” to a TCP port. In addition, broadcast packets may be sent — assuming the underlying network supports this — by using a reserved broadcast address; this address is network-interface dependent.

You can use options at the IP transport level with UDP (see the [IP](#) (p. 1391) protocol).

Based on:

RFC 768

Returns:

A descriptor referencing the socket, or -1 if an error occurs ([errno](#) (p. 539) is set).

Errors:

EADDRINUSE

You tried to create a socket with a port that has already been allocated.

EADDRNOTAVAIL

You tried to create a socket with a network address for which no network interface exists.

EISCONN

You tried to establish a connection on a socket that already has one, or to send a datagram with the destination address specified and the socket is already connected.

ENOBUFS

The system ran out of memory for an internal data structure.

ENOTCONN

You tried to send a datagram, but no destination address was specified and the socket hasn't been connected.

ultoa(), ulltoa()

Convert an unsigned long integer into a string, using a given base

Synopsis:

```
#include <stdlib.h>

char* ultoa( unsigned long int value,
             char* buffer,
             int radix );

char* ulltoa( unsigned long long value
             char* buffer,
             int radix );
```

Arguments:

value

The value to convert into a string.

buffer

A buffer in which the function stores the string. The size of the buffer must be at least 33 bytes when converting values in base 2 (binary).

radix

The base to use when converting the number. This value must be in the range:

2 *radix* 36

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ultoa()* and *ulltoa()* functions convert the unsigned binary integer *value* into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A NUL character is appended to the result.

Returns:

A pointer to the result.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

void print_value( unsigned long int value )
{
    int base;
    char buffer[33];

    for( base = 2; base <= 16; base = base + 2 )
        printf( "%2d %s\n", base,
                ultoa( value, buffer, base ) );
}

int main( void )
{
    print_value( (unsigned) 12765L );
    return EXIT_SUCCESS;
}
```

produces the output:

```
 2 11000111011101
 4 3013131
 6 135033
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

ultoa() is [QNX 4](#) (p. 46); *ulltoa()* is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

umask()

Set the file-mode creation mask for the process

Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask( mode_t cmask );
```

Arguments:

cmask

The new file-mode creation mask; that is, the permissions that you don't want set when the process creates a file. The mask is a combination of these bits:

Owner	Group	Others	Permission
S_IRUSR	S_IRGRP	S_IROTH	Read
S_IRWXU	S_IRWXG	S_IRWXO	Read, write, execute/search; a bitwise inclusive OR of the other three constants. (S_IRWXU is OR of IRUSR, S_IWSUR and S_IXUSR.)
S_IWUSR	S_IWGRP	S_IWOTH	Write
S_IXUSR	S_IXGRP	S_IXOTH	Execute/search

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `umask()` function sets the process's file-mode creation mask to `cmask`, and returns the previous value of the mask. Only the file permission bits (as defined in `<sys/stat.h>`) are used.



In order to change the mask for a process with a different effective user ID, your process must have the `PROCMGR_AID_UMASK` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The file-mode creation mask for the process is used when you call [creat\(\)](#) (p. 381), [mkdir\(\)](#) (p. 1636), [mkfifo\(\)](#) (p. 1639), and [open\(\)](#) (p. 1905), to turn off permission bits in the `mode` argument supplied. Bit positions set in `cmask` are cleared in the mode of the created file.

Returns:

The previous value of the file-mode creation mask.

Examples:

```
/*
 * Set the umask to RW for owner,group; R for other
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void )
{
    mode_t omask;
    mode_t nmask;

    nmask = S_IRUSR | S_IWUSR | /* owner read write */
           S_IRGRP | S_IWGRP | /* group read write */
           S_IROTH;           /* other read */
    omask = umask( nmask );
    printf( "Mask changed from %o to %o\n",
           omask, nmask );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

umount()

Unmount a filesystem

Synopsis:

```
#include <sys/mount.h>

int umount( const char* dir,
            int flags);
```

Arguments:

dir

The filesystem that you want to unmount.

flags

Flags that control the operation. Currently, the only valid value for *flags* is:

- `_MOUNT_FORCE` — force an unmount to occur.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `umount()` function sends a request to the server to unmount the path described by *dir*.

Returns:

-1 on failure ([errno](#) (p. 539) is set).

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

UNALIGNED_PUT16()

Write a misaligned 16-bit value safely

Synopsis:

```
#include <gulliver.h>

void UNALIGNED_PUT16( uint16_t *loc,
                      uint16_t num );
```

Arguments:

loc

The address where you want to write the value.

num

The value that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `UNALIGNED_PUT16()` macro lets you write the value *num* at the misaligned address *loc* without faulting.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

UNALIGNED_PUT16() is implemented as a macro.

UNALIGNED_PUT32()

Write a misaligned 32-bit value safely

Synopsis:

```
#include <gulliver.h>

void UNALIGNED_PUT32( uint32_t *loc,
                      uint32_t num );
```

Arguments:

loc

The address where you want to write the value.

num

The value that you want to write.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *UNALIGNED_PUT32()* macro lets you write the value *num* at the misaligned address *loc* without faulting.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

UNALIGNED_PUT32() is implemented as a macro.

UNALIGNED_PUT64()

Write a misaligned 64-bit value safely

Synopsis:

```
#include <gulliver.h>

void UNALIGNED_PUT64( uint64_t * loc,
                      uint64_t num );
```

Arguments:

loc

The address where you want to write the value.

num

The value that you want to write.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *UNALIGNED_PUT64()* macro lets you write the value *num* at the misaligned address *loc* without faulting.



This macro isn't currently implemented.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

UNALIGNED_PUT64() is implemented as a macro.

UNALIGNED_RET16()*Access a misaligned 16-bit value safely***Synopsis:**

```
#include <gulliver.h>

uint16_t UNALIGNED_RET16( const uint16_t *loc );
```

Arguments:***loc***

The address where you want to get the value from.

Library:**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *UNALIGNED_RET16()* macro lets you access the misaligned 16-bit value pointed to by *loc* without faulting.

Returns:The 16-bit value pointed to by *loc*.**Classification:**[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

UNALIGNED_RET16() is implemented as a macro.

UNALIGNED_RET32()

Access a misaligned 32-bit value safely

Synopsis:

```
#include <gulliver.h>

uint32_t UNALIGNED_RET32( const uint32_t *loc );
```

Arguments:

loc

The address where you want to get the value from.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *UNALIGNED_RET32()* macro lets you access the misaligned 32-bit value pointed to by *loc* without faulting.

Returns:

The 32-bit value pointed to by *loc*.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

UNALIGNED_RET32() is implemented as a macro.

UNALIGNED_RET64()

Access a misaligned 64-bit value safely

Synopsis:

```
#include <gulliver.h>

uint64_t UNALIGNED_RET64( const uint64_t * loc );
```

Arguments:

loc

The address where you want to get the value from.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *UNALIGNED_RET64()* macro lets you access the misaligned 64-bit value pointed to by *loc* without faulting.



This macro isn't currently implemented.

Returns:

The 64-bit value pointed to by *loc*.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

UNALIGNED_RET64() is implemented as a macro.

uname()

Get information about the operating system

Synopsis:

```
#include <sys/utsname.h>

int uname( struct utsname * name );
```

Arguments:

name

A pointer to a `utsname` where the function can store the information; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `uname()` function stores information about the current operating system in the structure pointed to by the argument *name*.

The system name structure, `utsname`, is defined in `<sys/utsname.h>`, and contains at least the following structure members:

`char* sysname`

The name of the OS.

`char* nodename`

The name of this node.

`char* release`

The current release level.

`char* version`

The current version level.

`char* machine`

The hardware type.

Each of these items is a null-terminated character array.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Examples:

```
/*
 * The following program prints some information about the
 * system it's running on.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utsname.h>

int main( void )
{
    struct utsname sysinfo;

    if( uname( &sysinfo ) == -1 ) {
        perror( "uname" );
        return EXIT_FAILURE;
    }
    printf( "system name   : %s\n", sysinfo.sysname );
    printf( "node name      : %s\n", sysinfo.nodename );
    printf( "release name   : %s\n", sysinfo.release );
    printf( "version name  : %s\n", sysinfo.version );
    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

ungetc()

Push a character back onto an input stream

Synopsis:

```
#include <stdio.h>

int ungetc( int c,
            FILE *fp );
```

Arguments:

c

The character that you want to push back.

fp

The stream you want to push the character back on.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ungetc()* function pushes the character specified by *c* back onto the input stream pointed to by *fp*. This character will be returned the next time that you read from the stream. The pushed-back character is discarded if you call *fflush()* (p. 641) or a file-positioning function (*fseek()* (p. 772), *fsetpos()* (p. 776), or *rewind()* (p. 2646)) before performing the next read operation.

Only one character (the most recent one) of pushback is guaranteed.

The *ungetc()* function clears the end-of-file indicator, unless the value of *c* is EOF.

Returns:

The character pushed back.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
int main( void )
{
    FILE *fp;
    int c;
    long value;

    fp = fopen( "file", "r" );
    value = 0;
    c = fgetc( fp );
    while( isdigit(c) ) {
        value = value*10 + c - '0';
        c = fgetc( fp );
    }
    ungetc( c, fp ); /* put last character back */
    printf( "Value=%ld\n", value );
    fclose( fp );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

ungetwc()

Push a wide character back onto an input stream

Synopsis:

```
#include <wchar.h>

wint_t ungetwc( wint_t wc,
                FILE * fp );
```

Arguments:

c

The wide character that you want to push back.

fp

The stream you want to push the wide character back on.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *ungetwc()* function pushes the wide character specified by *wc* back onto the input stream pointed to by *fp*.

The pushed-back character will be returned the next time that you read from the stream but is discarded if you call *fflush()* (p. 641) or a file-positioning function (*fseek()* (p. 772), *fsetpos()* (p. 776), or *rewind()* (p. 2646)) before the next read operation is performed.

Only one character (the most recent one) of pushback is guaranteed.

The *ungetwc()* function clears the end-of-file indicator, unless the value of *wc* is `WEOF`.

Returns:

The character pushed back.

Errors:

EILSEQ

Invalid character sequence or wide character.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

UNIX

UNIX-domain protocol family

Synopsis:

```
#include <sys/socket.h>
#include <sys/un.h>

socket( AF_LOCAL,
        SOCK_STREAM,
        0 );

socket( AF_LOCAL,
        SOCK_DGRAM,
        0 );
```

Description:

The UNIX-domain protocol family provides local (on-machine or QNX-network) interprocess communication through the normal *socket()* mechanisms. The UNIX-domain family supports the *SOCK_STREAM* and *SOCK_DGRAM* socket types and uses filesystem pathnames for addressing.

Addressing

UNIX-domain addresses are variable-length filesystem pathnames of at most 104 characters. The *<sys/un.h>* include file defines this address:

```
struct sockaddr_un {
    u_char  sun_len;
    u_char  sun_family;
    char    sun_path[104];
};
```

Binding a name to a UNIX-domain socket with *bind()* (p. 208) causes a socket file to be created in the filesystem. This file isn't removed when the socket is closed; you must use *unlink()* (p. 3579) to remove the file.

If the leading byte of the address is *'\0'*, the socket is an *abstract* Unix-domain socket. Abstract sockets don't create nodes in the path space, which in turn means they don't need to be unlinked.

You can use the macro *SUN_LEN()* (defined in *<sys/un.h>*) to calculate the length of UNIX-domain address, required by *bind()* and *connect()* (p. 348). The *sun_path* field must be terminated by a NUL character to be used with *SUN_LEN()*, but the terminating NUL isn't part of the address.

The UNIX-domain protocol family doesn't support broadcast addressing or any form of “wildcard” matching on incoming messages. All addresses are absolute- or relative-pathnames of other UNIX-domain sockets. Normal filesystem access-control

mechanisms are also applied when referencing pathnames (e.g. the destination of a *connect()* or *sendto()* (p. 2837) must be writable).

Protocols

The UNIX-domain protocol family consists of simple transport protocols that support the `SOCK_STREAM` and `SOCK_DGRAM` abstractions. UNIX-domain sockets also support the communication of QNX Neutrino file descriptors through the use of the *msg_control* field in the *msg* argument to *sendmsg()* (p. 2833) and *recvmsg()* (p. 2539).

Any valid descriptor may be sent in a message. The file descriptor to be passed is described using a `struct cmsghdr` defined in the include file `<sys/socket.h>`. The type of the message is `SCM_RIGHTS`, and the data portion of the messages is an array of integers representing the file descriptors to be passed. The number of descriptors being passed is defined by the length field of the message; the length field is the sum of the size of the header plus the size of the array of file descriptors.

The received descriptor is a duplicate of the sender's descriptor, as if it were created with a call to *dup()* (p. 483). Descriptors awaiting delivery or purposely not received are automatically closed by the system when the destination socket is closed.

LOCAL_CREDS

There is one socket-level option for *setsockopt()* (p. 2909) and *getsockopt()* (p. 962) available in the UNIX-domain. The `LOCAL_CREDS` option may be enabled on a `SOCK_DGRAM` or a `SOCK_STREAM` socket. This option provides a mechanism for the receiver to receive the credentials of the process as a *recvmsg()* message. The *msg_control* field in the `msghdr` structure points to a buffer that contains a `cmsghdr` structure followed by a variable length `sockcred` structure defined in `<sys/socket.h>` as follows:

```
struct sockcred {
    uid_t    sc_uid;           /* real user id */
    uid_t    sc_euid;          /* effective user id */
    gid_t    sc_gid;           /* real group id */
    gid_t    sc_egid;          /* effective group id */
    int      sc_ngroups;       /* number of supplemental groups */
    gid_t    sc_groups[1];     /* variable length */
};
```

The *SOCKCREDSIZE()* macro computes the size of the `sockcred` structure for a specified number of groups. The `cmsghdr` fields have the following values:

```
msg_len = sizeof(struct cmsghdr) + SOCKCREDSIZE(ngroups)
msg_level = SOL_SOCKET
msg_type = SCM_CREDS
```

unlink()

Remove a link to a file

Synopsis:

```
#include <unistd.h>

int unlink( const char * path );
```

Arguments:

path

The name of the file that you want to unlink.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *unlink()* function removes a link to a file:

- If the *path* names a symbolic link, *unlink()* removes the link, but doesn't affect the file or directory that the link goes to.
- If the *path* isn't a symbolic link, *unlink()* removes the link and decrements the link count of the file that the link refers to.

If the link count of the file becomes zero, and no process has the file open, then the space that the file occupies is freed, and no one can access the file anymore.

If one or more processes have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed.



To remove a directory, call [rmdir\(\)](#) (p. 2658) or [remove\(\)](#) (p. 2556).

Returns:

0

The operation was successful.

Nonzero

The operation failed (*errno* (p. 539) is set).

Errors:**EACCES**

Search permission is denied for a component of *path*, or write permission is denied on the directory containing the link to be removed.

EBUSY

The directory named by *path* cannot be unlinked because it's being used by the system or another process, and the target filesystem or resource manager considers this to be an error.

ENAMETOOLONG

The *path* argument exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.

ENOENT

The named file doesn't exist, or *path* is an empty string.

ENOSYS

The *unlink()* function isn't implemented for the filesystem specified by *path*.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The file named by *path* is a directory, and either the calling process doesn't have the appropriate privileges, or the target filesystem or resource manager prohibits using *unlink()* on directories.

EROFS

The directory entry to be unlinked resides on a read-only filesystem.

Examples:

```
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    if( unlink( "vm.tmp" ) ) {
        puts( "Error removing vm.tmp!" );
    }
}
```



```

        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

```

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

unlockpt()

Unlock a pseudo-terminal master/slave pair

Synopsis:

```
#include <stdlib.h>

int unlockpt( int fildes );
```

Arguments:

fildes

The file descriptor for the *master* pseudo-terminal.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *unlockpt()* function unlocks the slave pseudo-terminal device associated with the master to which *fildes* refers. Call *unlockpt()* before you open the slave side of a pseudo-terminal device.

Returns:

0, or -1 if an error occurred (*errno* (p. 539) is set).

Errors:

EBADF

The *fildes* argument isn't a file descriptor open for writing.

EINVAL

The *fildes* argument isn't associated with a master pseudo-terminal device.

Classification:*POSIX 1003.1 XSI* (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

unsetenv()

Remove an environment variable

Synopsis:

```
#include <stdlib.h>

int unsetenv( const char* name );
```

Arguments:

name

The name of the environment variable that you want to delete.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *unsetenv()* function removes the environment variable named *name* from the process's environment. If the named variable doesn't exist, *unsetenv()* indicates success and doesn't change the environment.

Returns:

0

Success.

-1

An error occurred; [errno](#) (p. 539) is set, and the environment isn't changed.

Errors:

EINVAL

The *name* argument is `NULL`, points to an empty string, or points to a string containing an equals (=) character.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The `unsetenv()` function manipulates the environment pointed to by the global [environ](#) (p. 528) variable.

usleep()

Suspend a thread for a given number of microseconds

Synopsis:

```
#include <unistd.h>

int usleep( useconds_t useconds );
```

Arguments:

useconds

The number of microseconds that you want to process to sleep for. This must be less than 1,000,000.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *usleep()* function suspends the calling thread until *useconds* microseconds of realtime have elapsed, or until a signal that isn't ignored is received. The time spent suspended could be longer than the requested amount due to the nature of time measurement (see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*), or due to the scheduling of other, higher-priority threads.

If *useconds* is 0, *usleep()* has no effect.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

EAGAIN

No timer resources are available to satisfy the request.

EINVAL

The *useconds* argument is too large.

Examples:

```
/*
 * The following program sleeps for the
 * number of microseconds specified in argv[1].
 */
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    useconds_t microseconds;

    microseconds = (useconds_t)strtol( argv[1], NULL, 0 );
    if( usleep( microseconds ) == 0 ) {
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

utime()

Record the modification time for a file or directory

Synopsis:

```
#include <sys/types.h>
#include <utime.h>

struct utimbuf {
    time_t  actime;    /* access time */
    time_t  modtime;   /* modification time */
};

int utime( const char* path,
           const struct utimbuf* times );
```

Arguments:

path

The path name for the file whose modification time you want to get or set.

times

NULL, or a pointer to a `utimbuf` structure where the function can store the modification time.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `utime()` function records the modification time for the file or directory identified by *path*.

If the *times* argument is NULL, the access and modification times of the file or directory are set to the current time. The effective user ID of the process must match the owner of the file or directory, or the process must have write permission to the file or directory, or appropriate privileges in order to use the `utime()` function in this way.

If the *times* argument isn't NULL, its interpreted as a pointer to a `utimbuf` structure, and the access and modification times of the file or directory are set to the values contained in the designated structure. Only the owner of the file or directory, and processes with appropriate privileges are permitted to use the `utime()` function in this

way. The access and modification times are taken from the *actime* and *modtime* fields in this structure.

Returns:

0

Success.

-1

An error occurred; *errno* is set.

Errors:

EACCES

Search permission is denied for a component of *path*, or the *times* argument is NULL, and the effective user ID of the process doesn't match the owner of the file, and write access is denied.

ENAMETOOLONG

The argument *path* exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.

ENOENT

The specified *path* doesn't exist, or *path* is an empty string.

ENOTDIR

A component of *path* isn't a directory.

EPERM

The *times* argument isn't NULL, and the calling process's effective user ID has write access to the file but doesn't match the owner of the file, and the calling process doesn't have the appropriate privileges.

EROFS

The named file resides on a read-only filesystem.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <utime.h>

int main( int argc, char *argv[] )
{
    if( (utime( argv[1], NULL ) != 0) && (argc > 1) ) {
        printf( "Unable to set time for %s\n", argv[1] );
    }
}
```

```
    return EXIT_SUCCESS;  
}
```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

utimes()

Set a file's access and modification times

Synopsis:

```
#include <sys/time.h>

int utimes( const char * __path,
            const struct timeval * __times );
```

Arguments:

__path

The name of the files whose times you want to set.

__times

NULL, or an array of `timeval` structures:

- The first array member represents the date and time of last access.
- The second member represents the date and time of last modification.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *utimes()* function sets the access and modification times of the file pointed to by the *__path* argument to the value of the *__times* argument. This function allows time specifications accurate to the microsecond.

The times in the `timeval` structure are measured in seconds and microseconds since the Unix Epoch (00:00:00 January 1, 1970 Coordinated Universal Time (UTC)), although rounding toward the nearest second may occur.

If the *__times* argument is `NULL`, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file, or must have write access to the file or superuser privileges to use this call in this manner. On completion, *utimes()* marks the time of the last file status change, *st_ctime*, for update.

Returns:

0

Success.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:**EACCES**

Search permission is denied by a component of the path prefix; or the `__times` argument is `NULL` and the effective user ID of the process doesn't match the owner of the file and write access is denied.

EFAULT

The `__path` or `__times` argument points to an illegal address.

EINTR

A signal was caught during the `utimes()` function.

EINVAL

The number of microseconds specified in one or both of the `timeval` structures pointed to by `__times` was greater than or equal to 1,000,000 or less than 0.

EIO

An I/O error occurred while reading from or writing to the filesystem.

ELOOP

Too many symbolic links were encountered in resolving `__path`.

EMULTIHOP

Components of `__path` require hopping to multiple remote machines and the filesystem doesn't allow it.

ENAMETOOLONG

The length of the `__path` argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.

ENOLINK

The `__path` argument points to a remote machine and the link to that machine is no longer active.

ENOENT

A component of `__path` doesn't name an existing file or `__path` is an empty string.

ENOTDIR

A component of the path prefix isn't a directory.

EPERM

The `__times` argument isn't `NULL` and the calling process's effective user ID has write access to the file but doesn't match the owner of the file, and the calling process doesn't have the appropriate privileges.

EROFS

The filesystem containing the file is read-only.

ENAMETOOLONG

Path name resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

utmp

Entry in a user-information file

Synopsis:

```
struct utmp {
    char    ut_user[UT_NAMESIZE];
#define ut_name ut_user
    char    ut_id[4];
    char    ut_line[UT_LINESIZE];
    pid_t   ut_pid;
    short   ut_type;
    struct exit_status {
        short e_termination;
        short e_exit;
    } ut_exit;
    short   ut_spare;
    time_t  ut_time;
};
```

Description:

The utmp structure describes an entry in a user-information file. The members include:

ut_user

The user's login name.

ut_id

The line number.

ut_line

The device name (console).

ut_pid

The process ID.

ut_type

The type of entry. The possible values are:

- EMPTY
- RUN_LVL
- BOOT_TIME
- OLD_TIME
- NEW_TIME

- INIT_PROCESS
- LOGIN_PROCESS
- USER_PROCESS
- DEAD_PROCESS
- ACCOUNTING

ut_exit

The exit status of a process marked as DEAD_PROCESS. The structure `exit_status` includes at least the following members:

- *e_termination* — the termination status.
- *e_exit* — the exit status.

ut_time

The time that this entry was made.

Classification:

[Unix](#) (p. 46)

utmpname()

Change the name of the user-information file

Synopsis:

```
#include <utmp.h>

void utmpname( char * __filename );
```

Arguments:

__filename

The new filename that you want to use.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *utmpname()* function lets you change the name of the file examined from the default file (`_PATH_UTMP`) to any other file. If the file doesn't exist, this won't be apparent until the first attempt to reference the file is made. This function doesn't open the file. It just closes the old file if it's currently open and saves the new file name.

Files:

_PATH_UTMP

Specifies the user information file.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

utoa()

Convert an unsigned integer into a string, using a given base

Synopsis:

```
#include <stdlib.h>

char* utoa( unsigned int value,
            char* buffer,
            int radix );
```

Arguments:

value

The value to convert into a string.

buffer

A buffer in which the function stores the string. The size of the buffer must be at least:

$8 \times \text{sizeof}(\text{int}) + 1$

bytes when converting values in base 2 (binary).

radix

The base to use when converting the number.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *utoa()* function converts the unsigned binary integer *value* into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A null character is appended to the result.

Returns:

A pointer to the result.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int base;
    char buffer[18];

    for( base = 2; base <= 16; base = base + 2 )
        printf( "%2d %s\n", base,
            utoa( (unsigned) 12765, buffer, base ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
2 11000111011101
4 3013131
6 135033
8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

[QNX 4](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Chapter 24

V

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “V”.

va_arg()

Get the next item in a list of variable arguments

Synopsis:

```
#include <stdarg.h>

type va_arg( va_list param,
             type );
```

Arguments:

param

The `va_list` object that you initialized with the [va_start\(\)](#) (p. 3608) macro.

type

The type of the next argument.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use the `va_arg()` macro to get the next argument in a list of variable arguments.

You must use `va_arg()` with the associated macros [va_copy\(\)](#) (p. 3605), [va_start\(\)](#) (p. 3608) and [va_end\(\)](#) (p. 3607). A sequence such as:

```
void example( char *dst, ... )
{
    va_list curr_arg;
    int next_arg;

    va_start( curr_arg, dst );
    next_arg = va_arg( curr_arg, int );
    ...
}
```

causes `next_arg` to be assigned the value of the next variable argument. The argument *type* (which is `int` in the example) is the type of the argument originally passed to the function.



The last argument before the ellipsis (`...`) has to be an `int` or a type that doesn't change in size if cast to an `int`. If the argument is promoted, the

ANSI/ISO standard says the behavior is undefined, and so depends on the compiler and the library.

You must execute the macro *va_start()* first, in order to initialize the variable *curr_arg* properly, and execute the macro *va_end()* after getting all the arguments.

The data item *curr_arg* is of type *va_list* that contains the information to permit successive acquisitions of the arguments.

The following functions use a “varargs” list:

- [verr\(\)](#) (p. 3612)
- [verrx\(\)](#) (p. 3612)
- [vfprintf\(\)](#) (p. 3616)
- [vfscanf\(\)](#) (p. 3618)
- [vfwprintf\(\)](#) (p. 3620)
- [vfwscanf\(\)](#) (p. 3622)
- [vprintf\(\)](#) (p. 3624)
- [vscanf\(\)](#) (p. 3626)
- [vslogf\(\)](#) (p. 3630)
- [vsprintf\(\)](#) (p. 3632)
- [vsscanf\(\)](#) (p. 3635)
- [vsscanf\(\)](#) (p. 3637)
- [vswprintf\(\)](#) (p. 3639)
- [vswscanf\(\)](#) (p. 3641)
- [vsyslog\(\)](#) (p. 3643)
- [vwarn\(\)](#) (p. 3645)
- [vwarnx\(\)](#) (p. 3645)
- [vwprintf\(\)](#) (p. 3647)
- [vwscanf\(\)](#) (p. 3649)

Returns:

The value of the next variable argument, according to type passed as the second parameter.

Examples:

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

static void test_fn( const char *msg,
                    const char *types,
                    ... );

int main( void )
{
    printf( "VA...TEST\n" );
    test_fn( "PARAMETERS: 1, \"abc\", 546",
```

```

        "isi", 1, "abc", 546 );
test_fn( "PARAMETERS: \"def\\", 789",
        "si", "def", 789 );
return EXIT_SUCCESS;
}

static void test_fn(
    const char *msg, /* message to be printed */
    const char *types, /* parameter types (i,s) */
    ... ) /* variable arguments */
{
    va_list argument;
    int arg_int;
    char *arg_string;
    const char *types_ptr;

    types_ptr = types;
    printf( "\\n%s -- %s\\n", msg, types );
    va_start( argument, types );
    while( *types_ptr != '\\0' ) {
        if ( *types_ptr == 'i' ) {
            arg_int = va_arg( argument, int );
            printf( "integer: %d\\n", arg_int );
        } else if ( *types_ptr == 's' ) {
            arg_string = va_arg( argument, char * );
            printf( "string: %s\\n", arg_string );
        }
        ++types_ptr;
    }
    va_end( argument );
}

```

produces the output:

VA...TEST

```

PARAMETERS: 1, "abc", 546 -- isi
integer: 1
string: abc
integer: 546

```

```

PARAMETERS: "def", 789 -- si
string: def
integer: 789

```

Classification:

[*POSIX 1003.1*](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

va_arg() is a macro.

va_copy()

Make a copy of a variable argument list

Synopsis:

```
#include <stdarg.h>

void va_copy( va_list d,
              va_list s );
```

Arguments:

d

A `va_list` object into which you want to copy the list.

s

The `va_list` object that you initialized with the [va_start\(\)](#) (p. 3608) macro and that you want to copy.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `va_copy()` macro creates a copy of a list of variable arguments.

You can use the `va_copy()` macro with the associated macros [va_arg\(\)](#) (p. 3602), [va_start\(\)](#) (p. 3608), and [va_end\(\)](#) (p. 3607).

Examples:

See [va_arg\(\)](#) (p. 3602).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

Caveats:

va_copy() is a macro.

va_end()

Finish getting items from a variable argument list

Synopsis:

```
#include <stdarg.h>

void va_end( va_list param );
```

Arguments:

param

The `va_list` object that you initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

Use the `va_end()` macro to complete the acquisition of arguments from a list of variable arguments. You must use it with the associated macros [va_copy\(\)](#) (p. 3605), [va_start\(\)](#) (p. 3608), and [va_arg\(\)](#) (p. 3602). For more information, see [va_arg\(\)](#).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

`va_end()` is a macro.

va_start()

Start getting items from a variable argument list

Synopsis:

```
#include <stdarg.h>

void va_start( va_list param,
               previous );
```

Arguments:

param

A `va_list` object that the “varargs” macros can use to locate the arguments.

previous

The argument that immediately precedes the “...” notation in the original function definition.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

Use the `va_start()` macro to start the acquisition of arguments from a list of variable arguments.

You must use the `va_start()` macro with the associated macros [va_arg\(\)](#) (p. 3602), [va_copy\(\)](#) (p. 3605), and [va_end\(\)](#) (p. 3607). For each call to `va_start()`, you must have a matching call to `va_end()`. For more information, see [va_arg\(\)](#).

Examples:

See [va_arg\(\)](#) (p. 3602).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

va_start() is a macro.

valloc()

Allocate a heap block aligned on a page boundary

Synopsis:

```
#include <stdarg.h>

void * valloc( size_t size);
```

Arguments:

size

The size of the block to allocate, in bytes.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `valloc()` function allocates a heap block that's aligned on a page boundary. It's equivalent to:

```
memalign( sysconf( _SC_PAGESIZE ), size );
```



Because the `malloc()` implementation uses signed, 32-bit integers to represent the size internally, you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, `valloc()` indicates an error of `ENOMEM`.

Returns:

A pointer to the allocated block, or `NULL` if an error occurred ([errno](#) (p. 539) is set).

Errors:

ENOMEM

There's insufficient memory available with the requested alignment.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

verr()*, *verrx()

Display a formatted error message, and then exit (varargs)

Synopsis:

```
#include <err.h>

void verr( int eval,
           const char *fmt,
           va_list args );

void verrx( int eval,
            const char *fmt,
            va_list args );
```

Arguments:

eval

The value to use as the exit code of the process.

fmt

NULL, or a [printf\(\)](#) (p. 2133)-style string used to format the message.

args

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The [err\(\)](#) and [warn\(\)](#) family of functions display a formatted error message on [stderr](#) (p. 3175). For a comparison of the members of this family, see [err\(\)](#) (p. 536).

The [verr\(\)](#) function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the *fmt* argument isn't NULL
- the string associated with the current value of [errno](#) (p. 539)

- a newline character.

The *verrx()* function produces a similar message, except that it doesn't include the string associated with *errno*. The message consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, if the *fmt* argument isn't NULL
- a newline character.

The *verr()* and *verrx()* functions don't return, but exit with the value of the argument *eval*.

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

vfork()

Spawn a new process and block the parent

Synopsis:

```
#include <process.h>

pid_t  vfork( void );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

This function spawns a new process and blocks the parent until the child process calls [execve\(\)](#) (p. 579) or exits (by calling [_exit\(\)](#) (p. 596) or abnormally).

Returns:

A value of zero to the child process, and (later) the child's process ID in the parent. If an error occurs, no child process is created, and the function returns -1 and sets [errno](#) (p. 539).

Errors:

EAGAIN

The system-imposed limit on the total number of processes under execution would be exceeded. This limit is determined when the system is generated.

ENOMEM

There isn't enough memory for the new process.

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	No

Caveats:

To avoid a possible deadlock situation, processes that are children in the middle of a *vfork()* are never sent **SIGTTOU** or **SIGTTIN** signals; rather, output or **ioctl**s are allowed and input attempts result in an **EOF** indication.

vfprintf()

Write formatted output to a file (varargs)

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>

int vfprintf( FILE* fp,
              const char* format,
              va_list arg );
```

Arguments:

fp

The stream to which you want to send the output.

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vfprintf()* function writes output to the file pointed to by *fp*, under control of the argument *format*.

The *vfprintf()* function is a “varargs” version of [fprintf\(\)](#) (p. 702).

Returns:

The number of characters written, or a negative value if an output error occurred ([errno](#) (p. 539) is set).

Examples:

```

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

FILE *LogFile;

/* a general error routine */
void errmsg( const char *format, ... )
{
    va_list arglist;

    fprintf( stderr, "Error: " );
    va_start( arglist, format );
    vfprintf( stderr, format, arglist );
    va_end( arglist );
    if( LogFile != NULL ) {
        fprintf( LogFile, "Error: " );
        va_start( arglist, format );
        vfprintf( LogFile, format, arglist );
        va_end( arglist );
    }
}

int main( void )
{
    LogFile = fopen( "error.log", "w" );
    errmsg( "%s %d %s", "Failed", 100, "times" );
    return EXIT_SUCCESS;
}

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

vfscanf()

Scan input from a file (varargs)

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>

int vfscanf( FILE *fp,
             const char *format,
             va_list arg );
```

Arguments:

fp

The stream that you want to read from.

format

A string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vfscanf()* function scans input from the file designated by *fp*, under control of the argument *format*.

The *vfscanf()* function is a “varargs” version of [fscanf\(\)](#) (p. 770).



The *vfscanf()* function is generally considered unsafe for string handling; it's safer to use [fgets\(\)](#) (p. 652) to get a line of input and then use [vsscanf\(\)](#) (p. 3637) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is stopped by reaching the end of the input stream before storing any values.

Errors:

If an error occurs, [errno](#) (p. 539) indicates the type of error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void ffind( FILE *fp, char *format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    vfprintf( fp, format, arglist );
    va_end( arglist );
}

int main( void )
{
    int day, year;
    char weekday[10], month[12];

    ffind( stdin,
           "%10s %12s %d %d",
           weekday, month, &day, &year );
    printf( "\n%s, %s %d, %d\n",
            weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

vfwprintf()

Write formatted wide-character output to a file (varargs)

Synopsis:

```
#include <wchar.h>
#include <stdarg.h>

int vfwprintf( FILE * fp,
               const wchar_t * format,
               va_list arg );
```

Arguments:

fp

The stream to which you want to send the output.

format

A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vfwprintf()* function writes output to the file pointed to by *fp*, under control of the argument *format*.

The *vfwprintf()* function is the wide-character version of [vfprintf\(\)](#) (p. 3616), and is a “varargs” version of [fwprintf\(\)](#) (p. 806).

Returns:

The number of wide characters written, excluding the terminating NUL, or a negative number if an error occurred ([errno](#) (p. 539) is set).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

vfwscanf()

Scan input from a file (varargs)

Synopsis:

```
#include <wchar.h>
#include <stdarg.h>

int vfwscanf( FILE * fp,
              const wchar_t *format,
              va_list arg );
```

Arguments:

fp

The stream that you want to read from.

format

A wide-character string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vfwscanf()* function scans input from the file designated by *fp*, under control of the argument *format*.

The *vfwscanf()* function is the wide-character version of [vscanf\(\)](#) (p. 3618), and is a “varargs” version of [fwscanf\(\)](#) (p. 810).



The *vfwscanf()* function is generally considered unsafe for string handling; it's safer to use [fgetws\(\)](#) (p. 658) to get a line of input and then use [vswscanf\(\)](#) (p. 3641) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

vprintf()

Write formatted output to standard output (varargs)

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>

int vprintf( const char* format,
             va_list arg );
```

Arguments:

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vprintf()* function writes output to the file *stdout*, under control of the argument *format*.

The *vprintf()* function is a “varargs” version of [printf\(\)](#) (p. 2133).

Returns:

The number of characters written, or a negative value if an output error occurred ([errno](#) (p. 539) is set).

Examples:

Use *vprintf()* in a general error message routine:

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <stdarg.h>

void errmsg( const char* format, ... )
{
    va_list arglist;

    printf( "Error: " );
    va_start( arglist, format );
    vprintf( format, arglist );
    va_end( arglist );
}

int main( void )
{
    errmsg( "%s %d %s", "Failed", 100, "times" );
    return EXIT_SUCCESS;
}

```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

vscanf()

Scan input from standard input (varargs)

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>

int vscanf( const char * format,
            va_list args );
```

Arguments:

format

A string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

args

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *vscanf()* function scans input from *stdin*, under control of the argument *format*. For information about the *format* string, see the description of [scanf\(\)](#) (p. 2713).

The *vscanf()* function is a “varargs” version of [scanf\(\)](#) (p. 2713).



The *vscanf()* function is generally considered unsafe for string handling; it's safer to use [fgets\(\)](#) (p. 652) to get a line of input and then use [sscanf\(\)](#) (p. 3162) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is stopped by reaching the end of the input stream before storing any values.

Errors:

If an error occurs, [errno](#) (p. 539) indicates the type of error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void ffind( char *format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    vscanf( format, arglist );
    va_end( arglist );
}

int main( void )
{
    int day, year;
    char weekday[10], month[12];

    ffind( "%10s %12s %d %d",
           weekday, month, &day, &year );
    printf( "\n%s, %s %d, %d\n",
           weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

vslog2f()

Log a printf-style string in a slog2 buffer (varargs)

Synopsis:

```
#include <slog2.h>

int vslog2f( slog2_buffer_t buffer,
             uint16_t code,
             uint8_t severity,
             const char* format,
             va_list arglist );
```

Arguments:

buffer

The handle of the buffer you want to log the message in, or NULL to use the default buffer that you specified earlier with [slog2_set_default_buffer\(\)](#) (p. 3057).

code

A user-specified code that you want to be associated with the message. The slog2 system doesn't interpret this code in any way.

severity

The severity level of this log item; one of:

- SLOG2_SHUTDOWN
- SLOG2_CRITICAL
- SLOG2_ERROR
- SLOG2_WARNING
- SLOG2_NOTICE
- SLOG2_INFO
- SLOG2_DEBUG1
- SLOG2_DEBUG2

For more information, see the entry for [slog2_register\(\)](#) (p. 3052).

format

A [printf](#) (p. 2133)-formatted character string used to define the log.

arglist

A `va_list` of the additional arguments required by the format string.

Library:

`lib slog2`

Use the `-l slog2` option to `qcc` to link against this library.

Description:

The `vslog2f()` function logs a printf-style ASCII-text string in the specified `slog2` buffer.

This function is a “varargs” version of [slog2f\(\)](#) (p. 3046).

Returns:

0 on success, or -1 if an error occurred.

Classification:

[QNX Neutrino](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

Caveats:

Don't use `double` or `float` arguments in an interrupt handler, unless your code is compiled to use software floating-point emulation.

vslogf()

Send a message to the system logger (varargs)

Synopsis:

```
#include <stdio.h>
#include <sys/slog.h>

int vslogf( int opcode,
            int severity,
            const char * fmt,
            va_list arg );
```

Arguments:

opcode

A combination of a *major* and *minor* code. Create the *opcode* using the `_SLOG_SETCODE( major, minor )` macro that's defined in `<sys/slog.h>`.

The *major* and *minor* codes are defined in `<sys/slogcodes.h>`.

severity

The severity of the log message; see “[Severity levels](#) (p. 3064),” in the documentation for *slogf()*.

fmt

A standard [printf\(\)](#) (p. 2133) string.

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The *slog*()* functions send log messages to the system logger, *slogger*. To send formatted messages, use *vslogf()*. If you have programs that scan log files for specified codes, you can use *slogb()* (p. 3061) or *slogi()* (p. 3066) to send a block of structures or *int*'s, respectively.

This function is a “varargs” version of *slogf()* (p. 3063).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

vsnprintf()

Write formatted output to a character array, up to a maximum number of characters (varargs)

Synopsis:

```
#include <stdarg.h>
#include <stdio.h>

int vsnprintf( char* buf,
               size_t count,
               const char* format,
               va_list arg );
```

Arguments:

buf

A pointer to the buffer where you want to function to store the formatted string.

count

The maximum number of characters to store in the buffer, including a terminating null character.

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vsnprintf()* function formats data under control of the *format* control string and stores the result in *buf*. The maximum number of characters to store, including a terminating null character, is specified by *count*.

The *vsnprintf()* function is a “varargs” version of [snprintf\(\)](#) (p. 3070).

Returns:

The number of characters that would have been written into the array, not counting the terminating null character, had *count* been large enough. It does this even if *count* is zero; in this case *buf* can be NULL.

If an error occurred, *vsnprintf()* returns a negative value and sets [errno](#) (p. 539).

Examples:

Use *vsnprintf()* in a general error message routine:

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

char msgbuf[80];

char *fmtmsg( char *format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    strcpy( msgbuf, "Error: " );
    vsnprintf( &msgbuf[7], 80-7, format, arglist );
    va_end( arglist );
    return( msgbuf );
}

int main( void )
{
    char *msg;

    msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
    printf( "%s\n", msg );

    return 0;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call *vsnprintf()* in a signal handler if the data isn't floating point.

vsprintf()

Write formatted output to a buffer (varargs)

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>

int vsprintf( char* buf,
              const char* format,
              va_list arg );
```

Arguments:

buf

A pointer to the buffer where you want the function to store the formatted string.

format

A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *vsprintf()* function formats data under control of the *format* control string, and writes the result to *buf*.

The *vsprintf()* function is a “varargs” version of [sprintf\(\)](#) (p. 3151).

Returns:

The number of characters written, or a negative value if an output error occurred ([errno](#) (p. 539) is set).

Examples:

Use *vsprintf()* in a general error message routine:

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>

char msgbuf[80];

char *fmtmsg( char *format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    strcpy( msgbuf, "Error: " );
    vsprintf( &msgbuf[7], format, arglist );
    va_end( arglist );
    return( msgbuf );
}

int main( void )
{
    char *msg;

    msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
    printf( "%s\n", msg );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call *vsprintf()* in a signal handler if the data isn't floating point.

vsscanf()

Scan input from a string (varargs)

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>

int vsscanf( const char* in_string,
             const char* format,
             va_list arg );
```

Arguments:

in_string

The string that you want to read from.

format

A string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vsscanf()` function scans input from the string designated by *in_string*, under control of the argument *format*.

The `vsscanf()` function is a “varargs” version of [sscanf\(\)](#) (p. 3162).

Returns:

The number of input arguments for which values were successfully scanned and stored is returned, or EOF when the scanning is terminated by reaching the end of the input string.

Examples:

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

void sfind( char* string, char* format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    vsscanf( string, format, arglist );
    va_end( arglist );
}

int main( void )
{
    int day, year;
    char weekday[10], month[12];

    sfind( "Monday June 28 1999",
           "%s %s %d %d",
           weekday, month, &day, &year );
    printf( "\n%s, %s %d, %d\n",
            weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call `vsscanf()` in a signal handler if the data isn't floating point.

vswprintf()

Write wide-character formatted output to a buffer (*varargs*)

Synopsis:

```
#include <wchar.h>
#include <stdarg.h>

int vswprintf( wchar_t * buf,
               size_t n,
               const wchar_t * format,
               va_list arg );
```

Arguments:

buf

A pointer to the buffer where you want the function to store the formatted string.

n

The maximum number of wide characters to store in the buffer, including a terminating null character.

format

A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `vswprintf()` function formats data under control of the *format* control string, and writes the result to *buf*.

The `vswprint()` function is the wide-character version of [vsprintf\(\)](#) (p. 3635), and is a “varargs” version of [swprintf\(\)](#) (p. 3272).

Returns:

The number of wide characters written, excluding the terminating NUL, or a negative number if an error occurred ([errno](#) (p. 539) is set).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call `vswprintf()` in a signal handler if the data isn't floating point.

vswscanf()

Scan input from a wide-character string (varargs)

Synopsis:

```
#include <wchar.h>
#include <stdarg.h>

int vswscanf( const wchar_t * ws,
              const wchar_t * format,
              va_list arg );
```

Arguments:

ws

The wide-character string that you want to read from.

format

A wide-character string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vswscanf()` function scans input from the string designated by `ws`, under control of the argument `format`.

The `vswscanf()` function is the wide-character version of [vsscanf\(\)](#) (p. 3637), and is a “varargs” version of [swscanf\(\)](#) (p. 3274).

Returns:

The number of input arguments for which values were successfully scanned and stored is returned, or EOF when the scanning is terminated by reaching the end of the input string.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Read the <i>Caveats</i>
Thread	Yes

Caveats:

It's safe to call `vswscanf()` in a signal handler if the data isn't floating point.

vsyslog()

Control system log (varargs)

Synopsis:

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog( int priority,
              const char *message,
              va_list args );
```

Arguments:

priority

The priority of the message; see “[Message levels](#) (p. 3316),” in the documentation for [syslog\(\)](#) (p. 3316).

message

The message that you want to write. This message is identical to a [printf\(\)](#) (p. 2133)-format string, except that %m is replaced by the current error message (as denoted by the global variable [errno](#) (p. 539)). A trailing newline is added if none is present.

args

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *vsyslog()* function writes *message* to the system message logger. The message is then written to the system console, log files, and logged-in users, or forwarded to other machines as appropriate. (See the `syslogd` command.)

This function is a “varargs” version of [syslog\(\)](#) (p. 3316).

Classification:*Unix* (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	No

vwarn(), vwarnx()

Formatted error message (varargs)

Synopsis:

```
#include <err.h>

void vwarn( const char *fmt,
            va_list args );

void vwarnx( const char *fmt,
             va_list args );
```

Arguments:

fmt

NULL, or a [printf\(\)](#) (p. 2133)-style string used to format the message.

args

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *err()* and *warn()* family of functions display a formatted error message on [stderr](#) (p. 3175). For a comparison of the members of this family, see [err\(\)](#) (p. 536).

The *vwarn()* function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the *fmt* argument isn't NULL
- the string associated with the current value of [errno](#) (p. 539)
- a newline character.

The *vwarnx()* function produces a similar message, except that it doesn't include the string associated with *errno*. The message consists of:

- the last component of the program name, followed by a colon and a space

- the formatted message, if the *fmt* argument isn't NULL
- a newline character.

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

vwprintf()

Write formatted output to standard output (varargs)

Synopsis:

```
#include <wchar.h>
#include <stdarg.h>

int vwprintf( const wchar_t * format,
              va_list arg );
```

Arguments:

format

A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *vwprintf()* function writes output to the file *stdout*, under control of the argument *format*.

The *vwprintf()* function is the wide-character version of [vprintf\(\)](#) (p. 3624), and is a “varargs” version of [wprintf\(\)](#) (p. 3736).

Returns:

The number of characters written, or a negative value if an output error occurred ([errno](#) (p. 539) is set).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

vwscanf()

Scan wide-character input from standard input (varargs)

Synopsis:

```
#include <wchar.h>
#include <stdarg.h>

int vwscanf( const wchar_t * format,
             va_list arg );
```

Arguments:

format

A wide-character string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

arg

A variable-argument list of the additional arguments, which you must have initialized with the [va_start\(\)](#) (p. 3608) macro.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `vwscanf()` function scans input from the file designated by `stdin`, under control of the argument *format*.

The `vwscanf()` function is the wide-character version of [vscanf\(\)](#) (p. 3626), and is a “varargs” version of [wscanf\(\)](#) (p. 3748).



The `vwscanf()` function is generally considered unsafe for string handling; it's safer to use [fgetws\(\)](#) (p. 658) to get a line of input and then use [vswscanf\(\)](#) (p. 3641) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 25

W

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

The following are described elsewhere:

For information about:	See:
<i>WCOREDUMP()</i>	wait() (p. 3652)
<i>WEXITSTATUS()</i>	wait() (p. 3652)
<i>WIFCONTINUED()</i>	wait() (p. 3652)
<i>WIFEXITED()</i>	wait() (p. 3652)
<i>WIFSIGNALED()</i>	wait() (p. 3652)
<i>WIFSTOPPED()</i>	wait() (p. 3652)
<i>WSTOPSIG()</i>	wait() (p. 3652)
<i>WTERMSIG()</i>	wait() (p. 3652)

This chapter describes the API components whose names begin with “W”.

wait()

Wait for the status of a terminated child process

Synopsis:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait( int * stat_loc );
```

Arguments:

stat_loc

NULL, or a pointer to a location where the function can store the terminating status of the child. For more information, see “[Status macros](#) (p. 3652),” below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wait()* function suspends execution of the calling thread until status information from one of its terminated child processes is available, or until the delivery of a signal whose action is either to terminate the process or execute a signal handler. If status information is available prior to the call to *wait()*, the return is immediate.



In order to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROCMGR_AID_WAIT` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

Status macros

If the *stat_loc* variable is non-NULL, the terminating status of the child process is in the location that it points to. The macros listed below, defined in `<sys/wait.h>`, extract information from *stat_loc*. The *stat_val* argument to these macros is the integer value pointed to by *stat_loc*.

POSIX defines the following macros:

WEXITSTATUS(stat_val)

Evaluates to the low-order 8 bits of the termination status of the child process if the value of *WIFEXITED(stat_val)* is nonzero.

WIFCONTINUED(stat_val)

Evaluates to a nonzero value if the status returned was from a child process that has continued from a job control stop.

WIFEXITED(stat_val)

Evaluates to a nonzero value if the status returned was from a normally terminated child process.

WIFSIGNALED(stat_val)

Evaluates to nonzero value if the child process terminated from reception of a signal that wasn't caught.

WIFSTOPPED(stat_val)

Evaluates to a nonzero value if the status returned is for a child process that's stopped.

WSTOPSIG(stat_val)

Evaluates to the number of the signal that caused the child process to stop if the value of *WIFSTOPPED(stat_val)* is nonzero.

WTERMSIG(stat_val)

Evaluates to the number of the signal that terminated the child process if the value of *WIFSIGNALED(stat_val)* is nonzero.

This macro isn't part of a POSIX standard:

WCOREDUMP(stat_val)

Evaluates to a nonzero value if the child process left a core dump.

One of the macros *WIFEXITED(*stat_loc)* and *WIFSIGNALED(*stat_loc)* evaluates to a nonzero value.

The non-POSIX [waitid\(\)](#) (p. 3663) function gives even more status information than the above macros.

Returns:

The process ID of the terminating child process, or -1 if an error occurred or on delivery of a signal ([errno](#) (p. 539) is set to EINTR).

Errors:**EINVAL**

The calling process has no existing unwaited-for child processes.

EINTR

The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wait3()

Wait for any child process to change its state

Synopsis:

```
#include <sys/wait.h>

pid_t wait3( int * stat_loc,
             int options,
             struct rusage * resource_usage );
```

Arguments:

stat_loc

NULL, or a pointer a location where the function can store the terminating status of the child process. For information about macros that extract information from this status, see “[Status macros](#) (p. 3652)” in the documentation for *wait()*.

options

A combination of zero or more of the following flags:

- **WCONTINUED** — return the status for any child that was stopped and has been continued.
- **WEXITED** — wait for the process(es) to exit.
- **WNOHANG** — return immediately if there are no children to wait for.
- **WNOWAIT** — keep the process in a waitable state. This doesn't affect the state of the process; the process may be waited for again after this call completion.
- **WSTOPPED** — wait for and return the process status of any child that has stopped because it received a signal.
- **WUNTRACED** — report the status of a stopped child process.

resource_usage

NULL, or a pointer to a `rusage` structure where the function can store information about resource usage. For information about this structure, see [getrusage\(\)](#) (p. 945).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wait3()` function allows the calling thread to obtain status information for specified child processes.



In order to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROC_MGR_AID_WAIT` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The following call:

```
wait3( stat_loc, options, resource_usage );
```

is equivalent to the call:

```
waitpid( (pid_t)-1, stat_loc, options );
```

except that on successful completion, if the `resource_usage` argument to `wait3()` isn't a null pointer, the `rusage` structure that the third argument points to is filled in for the child process identified by the return value.

It's also equivalent to:

```
wait4( (pid_t)-1, stat_loc, options, resource_usage );
```

Returns:

If the status of a child process is available, a value equal to the process ID of the child process for which status is reported.

If a signal is delivered to the calling process, `-1` and [errno](#) (p. 539) is set to `EINTR`.

Zero if `wait3()` is invoked with `WNOHANG` set in `options` and at least one child process is specified by `pid` for which status isn't available, and status isn't available for any process specified by `pid`.

Otherwise, `(pid_t)-1` and `errno` is set.

Errors:

ECHILD

The calling process has no existing unwaited-for child processes, or the set of processes specified by the argument `pid` can never be in the states specified by the argument `options`.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

New applications should use [waitpid\(\)](#) (p. 3666).

wait4()

Wait for one or more child process to change its state

Synopsis:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait4( pid_t pid,
             int * stat_loc,
             int options,
             struct rusage * resource_usage );
```

Arguments:

pid

The set of child processes that you want to get status information for:

- less than -1 — any child process whose process group ID is equal to the absolute value of *pid*.
- -1 — any child process
- 0 — any child process whose process group ID is equal to that of the calling process.
- greater than 0 — the single child process with this ID.

stat_loc

NULL, or a pointer a location where the function can store the terminating status of the child process. For information about macros that extract information from this status, see “[Status macros](#) (p. 3652)” in the documentation for *wait()*.

options

A combination of zero or more of the following flags:

- WCONTINUED — return the status for any child that was stopped and has been continued.
- WEXITED — wait for the process(es) to exit.
- WNOHANG — return immediately if there are no children to wait for.
- WNOWAIT — keep the process in a waitable state. This doesn't affect the state of the process; the process may be waited for again after this call completion.

- WSTOPPED — wait for and return the process status of any child that has stopped because it received a signal.
- WUNTRACED — report the status of a stopped child process.

resource_usage

NULL, or a pointer to a `rusage` structure where the function can store information about resource usage. For information about this structure, see [getrusage\(\)](#) (p. 945).

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wait4()` function suspends execution of the calling thread until status information from one of its terminated child processes is available, or until the delivery of a signal whose action is either to terminate the process or execute a signal handler. If status information is available prior to the call to `wait4()`, the return is immediate.



In order to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROCmgr_AID_WAIT` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The `wait4()` function behaves the same as the [wait\(\)](#) (p. 3652) function when passed a `pid` argument of `-1`, and the `options` argument has a value of zero.

Only one of the `WIFEXITED(stat_val)` and `WIFSIGNALED(stat_val)` macros can evaluate to a nonzero value.

The following call:

```
wait3( stat_loc, options, resource_usage );
```

is equivalent to the call:

```
waitpid( (pid_t)-1, stat_loc, options );
```

except that on successful completion, if the `resource_usage` argument to [wait3\(\)](#) (p. 3655) isn't a NULL pointer, the `rusage` structure that the third argument points to is filled in for the child process identified by the return value.

It's also equivalent to:

```
wait4( (pid_t)-1, stat_loc, options, resource_usage );
```

Returns:

If successful, *wait4()* returns the process ID of the terminating child process. If *wait4()* was invoked with *WNOHANG* set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, a value of zero is returned. On delivery of a signal *waitpid()* returns -1, and [errno](#) (p. 539) is set to *EINTR*.

Errors:**ECHILD**

The calling process has no existing unwaited-for child processes that meet the criteria set by *pid*.

EINTR

The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EINVAL

The value of the *options* argument isn't valid.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[Unix](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

waitfor()

Wait until a path exists

Synopsis:

```
#include <libgen.h>

int waitfor( const char *path,
             int delay_ms,
             int poll_ms );
```

Arguments:

path

The path you want to wait for.

delay_ms

The maximum time, in milliseconds, that you want to wait for.

poll_ms

The number of milliseconds to delay between checks.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *waitfor()* function waits for the given path to exist, up to a maximum time of *delay_ms* milliseconds, checking every *poll_ms* milliseconds. If you're waiting for a mountpoint, polling isn't necessary; you can specify 0 for *poll_ms* in this case.

Returns:

0

The path exists.

-1

An error occurred ([errno](#) (p. 539) is set).

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

waitid()

Wait for a child process to change state

Synopsis:

```
#include <sys/wait.h>

int waitid( idtype_t idtype,
            id_t id,
            siginfo_t * infop,
            int options );
```

Arguments:***idtype***

Which children you want to wait for:

- P_PID — the child with a process ID of (pid_t)*id*.
- P_PGID — any child with a process group ID equal to (pid_t)*id*.
- P_ALL — any child; *id* is ignored.

id

The process or process group ID that you want to wait for, depending on the value of *idtype*.

infop

A pointer to a siginfo_t structure, as defined in <sys/siginfo.h>, where the function can store the current state of the child; see below.

options

A combination of zero or more of the following flags:

- WCONTINUED — return the status for any child that was stopped and has been continued.
- WEXITED — wait for the process(es) to exit.
- WNOHANG — return immediately if there are no children to wait for.
- WNOWAIT — keep the process in a waitable state. This doesn't affect the state of the process; the process may be waited for again after this call completion.
- WSTOPPED — wait for and return the process status of any child that has stopped because it received a signal.

- **WUNTRACED** — report the status of a stopped child process.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `waitid()` function suspends the calling thread until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to `waitid()`, `waitid()` returns immediately.



In order to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROCmgr_AID_WAIT` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

If `waitid()` returns because a child process was found that satisfied the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* is filled in by the system with the status of the process. The *si_signo* member is always `SIGCHLD`.

If *idtype* is `P_ALL` and *options* is `WEXITED|WTRAPPED`, `waitid()` is equivalent to [wait\(\)](#) (p. 3652).

Returns:

0

One of the children changed its state. If `WNOHANG` was used, 0 can be returned (indicating no error); however, no children may have changed state if *info->si_pid* is 0.

-1

An error occurred ([errno](#) (p. 539) is set).

Errors:

ECHILD

The set of processes specified by *idtype* and *id* doesn't contain any unwaited-for processes.

EFAULT

The *infp* argument points to an illegal address.

EINTR

The *waitid()* function was interrupted by a signal.

EINVAL

An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

waitpid()

Wait for a child process to stop or terminate

Synopsis:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid( pid_t pid,
               int * stat_loc,
               int options );
```

Arguments:

pid

The set of child processes that you want to get status information for:

- less than -1 — any child process whose process group ID is equal to the absolute value of *pid*.
- -1 — any child process
- 0 — any child process whose process group ID is equal to that of the calling process.
- greater than 0 — the single child process with this ID.

stat_loc

NULL, or a pointer a location where the function can store the terminating status of the child process. For information about macros that extract information from this status, see “[Status macros](#) (p. 3652)” in the documentation for *wait()*.

options

A combination of zero or more of the following flags:

- WCONTINUED — return the status for any child that was stopped and has been continued.
- WEXITED — wait for the process(es) to exit.
- WNOHANG — return immediately if there are no children to wait for.
- WNOWAIT — keep the process in a waitable state. This doesn't affect the state of the process; the process may be waited for again after this call completion.

- **WSTOPPED** — wait for and return the process status of any child that has stopped because it received a signal.
- **WUNTRACED** — report the status of a stopped child process.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `waitpid()` function suspends execution of the calling thread until status information from one of its terminated child processes is available, or until the delivery of a signal whose action is either to terminate the process or execute a signal handler. If status information is available prior to the call to `waitpid()`, the return is immediate.



In order to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID, your process must have the `PROCmgr_AID_WAIT` ability enabled. For more information, see [procmgr_ability\(\)](#) (p. 2141).

The `waitpid()` function behaves the same as [wait\(\)](#) (p. 3652) when passed a `pid` argument of `-1`, and the `options` argument has a value of zero.

Only one of the `WIFEXITED(stat_val)` and `WIFSIGNALED(stat_val)` macros can evaluate to a nonzero value.

Returns:

The process ID of the terminating child process on success. If `waitpid()` is invoked with `WNOHANG` set in `options`, it has at least one child process specified by `pid` for which status isn't available, and status isn't available for any process specified by `pid`, a value of zero is returned. On delivery of a signal, `waitpid()` returns `-1`, and [errno](#) (p. 539) is set to `EINTR`.

Errors:**ECHILD**

The calling process has no existing unwaited-for child processes that meet the criteria set by `pid`.

EINTR

The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EINVAL

The value of the *options* argument isn't valid.

EPERM

The calling process doesn't have the required permission; see [procmgr_ability\(\)](#) (p. 2141).

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

warn(), warnx()

Formatted error message

Synopsis:

```
#include <err.h>

void warn( const char* fmt, ...);
void warnx( const char* fmt, ...);
```

Arguments:

fmt

NULL, or a [printf\(\)](#) (p. 2133)-style string used to format the message.

Additional arguments

As required by the format string.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *err()* and *warn()* family of functions display a formatted error message on [stderr](#) (p. 3175). For a comparison of the members of this family, see [err\(\)](#) (p. 536).

The *warn()* function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the *fmt* argument isn't NULL
- the string associated with the current value of [errno](#) (p. 539)
- a newline character.

The *warnx()* function produces a similar message, except that it doesn't include the string associated with *errno*. The message consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, if the *fmt* argument isn't NULL
- a newline character.

Examples:

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
    warnx("%s: %s: trying the block device",
        raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
    warn("%s", block_device);
```

Classification:

Unix (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wrtomb()

Convert a wide-character code to a character

Synopsis:

```
#include <wchar.h>

size_t wrtomb( char * s,
               wchar_t wc,
               mbstate_t * ps);
```

Arguments:

s

NULL, or a pointer to a location where the function can store the multibyte character.

wc

The wide character that you want to convert.

ps

An internal pointer that lets *wrtomb()* be a restartable version of [wctomb\(\)](#) (p. 3718); if *ps* is NULL, *wrtomb()* uses its own internal variable.

You can call [mbsinit\(\)](#) (p. 1589) to determine the status of this variable.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wrtomb()* function determines the number of bytes needed to represent the wide character *wc* as a multibyte character and stores the multibyte character in the location pointed to by *s*, to a maximum of MB_CUR_MAX bytes.

This function is affected by LC_CTYPE.

Returns:

The number of bytes stored, or (size_t)-1 if the variable *wc* is an invalid wide-character code.

Errors:**EILSEQ**

Invalid wide-character code.

EINVAL

The variable *ps* points to an invalid conversion state.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wcscat()

Concatenate two wide-character strings

Synopsis:

```
#include <wchar.h>

wchar_t * wcscat( wchar_t * ws1,
                  const wchar_t * ws2 );
```

Arguments:

ws1, ws2

The wide-character strings that you want to concatenate.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wcscat()* function appends a copy of the string pointed to by *ws2*, including the terminating NUL wide character, to the end of the string pointed to by *ws1*. The first wide character of *ws2* overwrites the NUL wide character at the end of *ws1*.

Returns:

The same pointer as *ws1*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcschr()

Find the first occurrence of a wide character in a string

Synopsis:

```
#include <wchar.h>

wchar_t * wcschr( const wchar_t * ws,
                  wchar_t wc );
```

Arguments:

- ws**
The wide-character string that you want to search.
- wc**
The wide character that you're looking for.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wcschr()* function finds the first occurrence of *wc* in the string pointed to by *ws*. The terminating NUL character is considered to be part of the string.

Returns:

A pointer to the located wide character, or NULL if *wc* doesn't occur in the string.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

wscmp()

Compare two wide-character strings

Synopsis:

```
#include <wchar.h>

int wscmp( const wchar_t * ws1,
           const wchar_t * ws2 );
```

Arguments:

ws1, ws2

The wide-character strings that you want to compare.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *wscmp()* function compares the wide-characters strings pointed to by *ws1* and *ws2*.

Returns:

< 0

ws1 is less than *ws2*.

0

ws1 is equal to *ws2*.

> 0

ws1 is greater than *ws2*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcscoll()

Compare two wide-character strings, using the locale's collating sequence

Synopsis:

```
#include <wchar.h>

int wcscoll( const wchar_t * ws1,
             const wchar_t * ws2 );
```

Arguments:

ws1, ws2

The wide-character strings that you want to compare.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *wcscoll()* function compares the wide-character strings pointed to by *ws1* and *ws2*, using the LC_COLLATE collating sequence selected by the [setlocale\(\)](#) (p. 2880) function.

Returns:

< 0

ws1 is less than *ws2*.

0

ws1 is equal to *ws2*.

> 0

ws1 is greater than *ws2*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	No

wcscpy()

Copy a wide-character string

Synopsis:

```
#include <wchar.h>

wchar_t * wcscpy( wchar_t * ws1,
                  const wchar_t * ws2 );
```

Arguments:

ws1

A pointer to where you want to copy the string.

ws2

The wide-character string that you want to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcscpy()` function copies the string pointed to by `ws2`, including the terminating NUL wide character, into the array pointed to by `ws1`.



This function isn't guaranteed to work properly for copying overlapping strings; use [wmemmove\(\)](#) (p. 3730) instead.

Returns:

The same pointer as `ws1`.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcscspn()

Count the wide characters at the beginning of a string that aren't in a given character set

Synopsis:

```
#include <wchar.h>

size_t wcscspn( const wchar_t * ws1,
                const wchar_t * ws2 );
```

Arguments:

- ws1**
The wide-character string that you want to search.
- ws2**
The set of wide characters you want to look for.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *strspn()* function returns the length of the initial segment of the string pointed to by *ws1* consisting entirely of wide characters *not* from the string pointed to by *ws2*. The terminating NUL isn't considered to be part of *ws2*.

Returns:

The length of the segment.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

wcsftime()

Format the time into a wide-character string

Synopsis:

```
#include <wchar.h>

size_t wcsftime( wchar_t * wcs,
                 size_t maxsize,
                 const wchar_t * format,
                 const struct tm * timeptr );
```

Arguments:

wcs

A pointer to a buffer where the function can store the formatted time.

maxsize

The maximum size of the buffer.

format

The format that you want to use for the time; see “[Formats](#) (p. 3202),” in the description of *strftime()*.

timeptr

A pointer to a [tm](#) (p. 3476) structure that contains the time that you want to format.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wcsftime()* function is similar to [strftime\(\)](#) (p. 3202), except that *wcsftime()* works with wide characters.

Returns:

The number of wide characters placed into the array, not including the terminating null character, or 0 if the number of wide characters exceeds *maxsize* (in this case, the string contents are indeterminate).

If an error occurs, [errno](#) (p. 539) is set.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wcslen()

Compute the length of a wide-character string

Synopsis:

```
#include <wchar.h>

size_t wcslen( const wchar_t * ws );
```

Arguments:

ws

The wide-character string whose length you want to calculate.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *wcslen()* function counts the wide characters in the string pointed to by *ws*.

Returns:

The number of wide characters, not counting the terminating NUL.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcsncat()

Concatenate two wide-character strings, up to a maximum length

Synopsis:

```
#include <wchar.h>

wchar_t * wcsncat( wchar_t * ws1,
                   const wchar_t * ws2
                   size_t n );
```

Arguments:

ws1, ws2

The wide-character strings that you want to concatenate.

n

The maximum number of wide characters that you want to add from the *ws2* string.

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `wcsncat()` function appends a copy of the string pointed to by *ws2*, including the terminating NUL wide character, to the end of the string pointed to by *ws1*. The first character of *ws2* overwrites the NUL wide character at the end of *ws1*. The function writes no more than *n* wide characters from *ws2* and appends a NUL wide character to the result.

Returns:

The same pointer as *ws1*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcsncmp()

Compare two wide-character strings, up to a given length

Synopsis:

```
#include <wchar.h>

int wcsncmp( const wchar_t * ws1,
             const wchar_t * ws2,
             size_t n );
```

Arguments:

ws1, ws2

The wide-character strings that you want to compare.

n

The maximum number of wide characters that you want to compare.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *wcsncmp()* function compares up to *n* wide characters from the strings pointed to by *ws1* and *ws2*.

Returns:

< 0

ws1 is less than *ws2*.

0

ws1 is equal to *ws2*.

> 0

ws1 is greater than *ws2*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcsncpy()

Copy a wide-character string, to a maximum length

Synopsis:

```
#include <wchar.h>

wchar_t * wcsncpy( wchar_t * ws1,
                   const wchar_t * ws2,
                   size_t n );
```

Arguments:

ws1

A pointer to where you want to copy the wide-character string.

ws2

The wide-character string that you want to copy.

n

The maximum number of wide characters that you want to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsncpy()` function copies the string pointed to by `ws2`, including the terminating NUL wide character, into the array pointed to by `ws1`, to a maximum of `n` wide characters. It adds NUL characters if `ws2` has fewer than `n` characters but doesn't add a NUL if `ws2` has more.



This function isn't guaranteed to work properly for copying overlapping strings; use [wmemmove\(\)](#) (p. 3730) instead.

Returns:

The same pointer as `ws1`.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcspbrk()

Find the first wide character in a string that's in a given character set

Synopsis:

```
#include <wchar.h>

wchar_t * wcspbrk( const wchar_t * ws1,
                   const wchar_t * ws2 );
```

Arguments:

- ws1**
The wide-character string that you want to search.
- ws2**
The set of wide characters you want to look for.

Library:

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wcspbrk()* function locates the first occurrence in the string pointed to by *ws1* of any wide character from the string pointed to by *ws2*.

Returns:

A pointer to the located character, or NULL if no character from *ws2* occurs in *ws1*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

wcsrchr()

Find the last occurrence of a wide character in a string

Synopsis:

```
#include <wchar.h>

wchar_t * wcsrchr( const wchar_t * ws,
                   wchar_t wc );
```

Arguments:

ws

The wide-character string that you want to search.

wc

The wide character that you're looking for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsrchr()` function finds the last occurrence of `wc` in the string pointed to by `ws`. The terminating NUL character is considered to be part of the string.

Returns:

A pointer to the located wide character, or `NULL` if `wc` doesn't occur in the string.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes

Safety:	
Thread	Yes

wcsrtombs()

Convert a wide-character string into a multibyte character string (restartable)

Synopsis:

```
#include <wchar.h>

size_t wcsrtombs( char * dst,
                  const wchar_t ** src,
                  size_t len,
                  mbstate_t * ps);
```

Arguments:

dst

A pointer to a buffer where the function can store the multibyte-character string.

src

A pointer to the wide-character string that you want to convert.

len

The maximum number of multibyte characters to store.

ps

An internal pointer that lets *wcsrtombs()* be a restartable version of [wcstombs\(\)](#) (p. 3711); if *ps* is NULL, *wcsrtombs()* uses its own internal variable.

You can call [mbsinit\(\)](#) (p. 1589) to determine the status of this variable.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wcsrtombs()* function converts a string of wide-characters pointed to by *src* into the corresponding multi-byte characters pointed to by *dst*. No more than *len* bytes are stored, including the terminating NULL character.

The function converts each character as if by a call to [wctomb\(\)](#) (p. 3718) and stops early if:

- A sequence of bytes doesn't conform to a valid character.
- Converting the next character would exceed the limit of *len* total bytes.

The *wcsrtombs()* function uses *ps* to make it thread safe; if *ps* is a NULL pointer, *wcsrtombs()* uses its own internal pointer.

Returns:

The number of total bytes successfully converted, not including the terminating NULL byte, or (size_t)-1 if an invalid wide-character code was found.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wcsspnp()

Count the wide characters at the beginning of a string that are in a given character set

Synopsis:

```
#include <wchar.h>

size_t wcsspnp( const wchar_t * ws1,
                const wchar_t * ws2 );
```

Arguments:

- ws1*
- The wide-character string that you want to search.
- ws2*
- The set of wide characters you want to look for.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wcsspnp()* function returns the length of the initial segment of the string pointed to by *ws1* consisting entirely of wide characters from the string pointed to by *ws2*. The terminating NUL isn't considered to be part of *ws2*.

Returns:

The length of the segment.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

wcsstr()

Find one wide-character string inside another

Synopsis:

```
#include <wchar.h>

wchar_t * wcsstr( const wchar_t * ws1,
                  const wchar_t * ws2 );
```

Arguments:

ws1

The wide-character string that you want to search.

ws2

The wide-character string that you're looking for.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsstr()` function locates the first occurrence in the string pointed to by `ws1` of the sequence of wide characters, excluding the terminating NUL, in the string pointed to by `ws2`.

Returns:

A pointer to the located string, NULL if the string wasn't found, or the same pointer as `ws1` if `ws2` points to a zero-length string.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes

Safety:	
Signal handler	Yes
Thread	Yes

wcstod(), wcstof(), wcstold()

Convert a wide-character string into a double, float, or long double

Synopsis:

```
#include <wchar.h>

double wcstod( const wchar_t * ptr,
               wchar_t ** endptr );

float wcstof( const wchar_t * ptr,
              wchar_t ** endptr );

long double wcstold( const wchar_t * ptr,
                    wchar_t ** endptr );
```

Arguments:

nptr

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

These functions convert a wide-character string to a number:

- *wcstod()* function converts it to a double
- *wcstof()* converts it to a float
- *wcstold()* to a long double.

These functions recognize strings containing the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits containing an optional decimal point
- an optional e or E, followed by an optionally signed sequence of digits.

The functions expect the string to have a plus or minus sign, followed by one of these forms:

- A sequence of decimal digits, optionally followed by a radix character, optionally followed by an exponent part.
- A 0x or 0X followed by a sequence of hexadecimal digits, optionally followed by a radix character, optionally followed by a binary exponent part.
- The case-insensitive string INF or INFINITY.
- The case-insensitive string NAN or NAN(*n-wchar-sequence*) where *n-wchar-sequence* may be a digit, a nondigit, a n-wchar-sequence digit or a n-wchar-sequence nondigit.

The value is correctly rounded if the subject is hexadecimal and FLT_RADIX is 2.

The radix character is locale specific, depending upon LC_NUMERIC.

The conversion ends at the first unrecognized character. If *endptr* isn't NULL, a pointer to the unrecognized wide character is stored in the object *endptr* points to.



Because 0 is a valid return that is also used for an error, you should set [errno](#) (p. 539) to 0 before calling these functions, and check *errno* again afterward. These functions don't change *errno* on success.

Returns:

The converted value. If the correct value would cause overflow, plus or minus HUGE_VAL is returned according to the sign, and *errno* is set to ERANGE. If the correct value would cause underflow, then zero is returned, and *errno* is set to ERANGE.

Zero is returned when the input string can't be converted. When an error occurs, *errno* indicates the error detected.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wcstoimax(), wcstoumax()

Convert a wide-character string into an integer

Synopsis:

```
#include <inttypes.h>

intmax_t wcstoimax ( const wchar_t * nptr,
                    wchar_t ** endptr,
                    int base );

uintmax_t wcstoumax ( const wchar_t * nptr,
                     wchar_t ** endptr,
                     int base );
```

Arguments:***nptr***

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

base

The base of the number being parsed:

- If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
- If *base* isn't zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The `wcstoimax()` and `wcstoumax()` functions are the same as the [wcstol\(\)](#), [wcstoll\(\)](#) (p. 3709), [wcstoul\(\)](#), and [wcstoull\(\)](#) (p. 3713) functions except that they return objects of type `intmax_t` and `uintmax_t`.

Returns:

The converted value.

If the correct value causes an overflow, (`INTMAX_MAX|UINTMAX_MAX` or `INTMAX_MIN`) is returned according to the sign and [errno](#) (p. 539) is set to `ERANGE`. If *base* is out of range, zero is returned and *errno* is set to `EINVAL`.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcstok()

Break a wide-character string into tokens

Synopsis:

```
#include <wchar.h>

wchar_t * wcstok( wchar_t * ws1,
                  const wchar_t * ws2,
                  wchar_t ** ptr );
```

Arguments:

ws1

NULL, or the wide-character string that you want to break into tokens; see below.

ws2

A set of the wide characters that separate the tokens.

ptr

The address of a pointer to a `wchar_t` object, which the function can use to store information necessary for it to continue scanning the same string.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `wcstok()` breaks the wide-character string pointed to by `ws1` into a sequence of tokens, each of which is delimited by a wide character from the string pointed to by `ws2`.

In the first call to `wcstok()`, `ws1` must point to a null-terminated string, `ws2` must point to a null-terminated string of separator wide characters, and `ptr` is ignored. The `wcstok()` function returns a pointer to the first wide character of the first token, writes a NUL wide character into `ws1` immediately following the returned token, and updates `ptr`.

In subsequent calls, `ws1` must be NULL, and `ptr` must be unchanged from the previous call so that subsequent calls will move through the string `ws1`, returning successive

tokens until no tokens remain. The separator string *ws2* may differ from call to call. When no tokens remain in *ws1*, a NULL pointer is returned.

Returns:

A pointer to the token found, or NULL if no token was found.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wcstol(), wcstoll()

Convert a wide-character string into a long integer

Synopsis:

```
#include <wchar.h>

long wcstol( const wchar_t * ptr,
             wchar_t ** endptr,
             int base );

long long wcstoll( const wchar_t * ptr,
                  wchar_t ** endptr,
                  int base );
```

Arguments:***ptr***

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

base

The base of the number being parsed:

- If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
- If *base* isn't zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wcstol()* function converts the string pointed to by *ptr* into a `long`; *wcstoll()* converts the string into a `long long`.

These functions recognize strings that contain the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits and letters.

The conversion ends at the first unrecognized wide character. If *endptr* isn't `NULL`, a pointer to the unrecognized wide character is stored in the object *endptr* points to.

Returns:

The converted value.

If the correct value causes an overflow, `LONG_MAX|LONGLONG_MAX` or `LONG_MIN|LONGLONG_MIN` is returned according to the sign, and [errno](#) (p. 539) is set to `ERANGE`. If *base* is out of range, zero is returned and *errno* is set to `EDOM`.

Errors:**ERANGE**

The value is not representable

EINVAL

The value for *base* is not supported or no conversion could be performed.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcstombs()

Convert a wide-character string into a multibyte character string

Synopsis:

```
#include <stdlib.h>

size_t wcstombs( char* s,
                  const wchar_t* pwcs,
                  size_t n );
```

Arguments:

s

A pointer to a buffer where the function can store the multibyte-character string.

pwcs

The wide-character string that you want to convert.

n

The maximum number of bytes to store.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The `wcstombs()` function converts a sequence of wide character codes from the array pointed to by `pwcs` into a sequence of multibyte characters, and stores them in the array pointed to by `s`. It stops if a multibyte character exceeds the limit of `n` total bytes, or if the NUL character is stored. At most `n` bytes of the array pointed to by `s` are modified.

The [wcsrtombs\(\)](#) (p. 3697) function is a restartable version of `wcstombs()`.

Returns:

The number of array elements modified, not including the terminating zero code, if present, or `(size_t)-1` if an invalid multibyte character is encountered.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

wchar_t wbuffer[] = {
    0x0073,
    0x0074,
    0x0072,
    0x0069,
    0x006e,
    0x0067,
    0x0000
};

int main( void )
{
    char    mbsbuffer[50];
    int     i, len;

    len = wcstombs( mbsbuffer, wbuffer, 50 );
    if( len != -1 ) {
        for( i = 0; i < len; i++ )
            printf( "%4.4x", wbuffer[i] );
        printf( "\n" );
        mbsbuffer[len] = '\0';
        printf( "%s(%d)\n", mbsbuffer, len );
    }
    return EXIT_SUCCESS;
}
```

produces the output:

```
/0073/0074/0072/0069/006e/0067
string(6)
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wcstoul(), wcstoull()

Convert a wide-character string into an unsigned long integer

Synopsis:

```
#include <wchar.h>

unsigned long wcstoul( const wchar_t * ptr,
                      wchar_t ** endptr,
                      int base );

unsigned long long wcstoull( const wchar_t * ptr,
                             wchar_t ** endptr,
                             int base );
```

Arguments:***ptr***

A pointer to the string to parse.

endptr

If this argument isn't NULL, the function stores in it a pointer to the first unrecognized character found in the string.

base

The base of the number being parsed:

- If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
- If *base* isn't zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

These functions convert a wide-character string into a number:

- `wcstoul()` converts the string into an unsigned long
- `wcstoull()` converts it into a unsigned long long.

These functions recognize a string containing optional white space, followed by a sequence of digits and letters. The conversion ends at the first unrecognized character. A pointer to that character is stored in the object `endptr` points to, if `endptr` isn't NULL.

If *base* is zero, the first characters determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.

If *base* isn't zero, it must have a value of between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Returns:

The converted value.

If the correct value causes an overflow, `ULONG_MAX/ULONGLONG_MAX` is returned and *errno* is set to `ERANGE`. If *base* is out of range, zero is returned and *errno* is set to `EDOM`.

Errors:

ERANGE

The value is not representable

EINVAL

The value for *base* is not supported or no conversion could be performed.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wcsxfrm()

Transform one wide-character string into another, to a given length

Synopsis:

```
#include <wchar.h>

size_t wcsxfrm( wchar_t * ws1,
                const wchar_t * ws2,
                size_t n );
```

Arguments:

ws1

The string that you want to transform.

ws2

The string that you want to place in *dst*.

n

The maximum number of characters to transform.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsxfrm()` function transforms the string pointed to by *ws2* to the buffer pointed to by *ws1*, to a maximum of *n* wide-characters, including the terminating null. The two strings shouldn't overlap.

A call to [wcscmp\(\)](#) (p. 3676) returns the same result for two strings transformed by `wcsxfrm()` as [wcscoll\(\)](#) (p. 3678) would return for the original versions of the strings.



This function doesn't report errors in its returns; set [errno](#) (p. 539) to 0, call `wcsxfrm()`, and then check `errno` again.

Returns:

The length of the transformed wide-character string. If this value is greater than n , the contents of `ws1` are indeterminate.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wctob()

Convert a wide character to a single-byte code

Synopsis:

```
#include <wchar.h>

int wctob( wint_t c );
```

Arguments:

c

The wide character that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wctob()` function returns the single-byte representation of a wide character.

This function is affected by `LC_CTYPE`.

Returns:

The single-byte representation, or EOF if *c* isn't a valid single-byte character.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wctomb()

Convert a wide character into a multibyte character

Synopsis:

```
#include <stdlib.h>
int wctomb( char * s,
            wchar_t wc );
```

Arguments:

s

NULL, or a pointer to a location where the function can store the multibyte character.

wc

The wide character that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wctomb()` function determines the number of bytes required to represent the multibyte character corresponding to the code contained in `wc`. If `s` isn't NULL, the multibyte character representation is stored in the array it points to. At most MB_CUR_MAX characters are stored.

Returns:

- If `s` is NULL:

0

The `wctomb()` function uses locale specific multibyte character encoding that's not state-dependent.

>0

The function is state-dependent.

- If *s* isn't NULL:

-1

If the value of *wchar* doesn't correspond to a valid multibyte character.

x

The number of bytes that comprise the multibyte character corresponding to the value of *wchar*.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

wchar_t wchar = { 0x0073 };
char mbbuffer[MB_CUR_MAX];

int main( void )
{
    int len;

    printf( "Character encodings do %shave "
           "state-dependent \nencoding.\n",
           ( wctomb( NULL, 0 ) )
           ? "" : "not " );

    len = wctomb( mbbuffer, wchar );
    mbbuffer[len] = '\\0';
    printf( "%s(%d)\n", mbbuffer, len );
    return EXIT_SUCCESS;
}
```

This produces the output:

```
Character encodings do not have state-dependent
encoding.
s(1)
```

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wctrans()

Define a wide-character mapping

Synopsis:

```
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

Arguments:

property

The type of mapping; see below.

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wctrans()* function determines a mapping rule for wide-character codes according to the category LC_CTYPE, particularly for use with *towctrans()* (p. 3486).

The following mappings are defined in all locales, although additional classes may be defined for LC_CTYPE:

- tolower
- toupper

Use *setlocale()* (p. 2880) to modify the category LC_CTYPE.

Returns:

An object that you can use in a call to *towctrans()* (p. 3486), or 0 if the specified character mapping isn't valid for the current locale.

Classification:

ANSI (p. 46), *POSIX 1003.1* (p. 46)

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wctype()

Define a wide-character class

Synopsis:

```
#include <wctype.h>

wctype_t wctype( const char * property );
```

Arguments:

property

A string that defines the property of the class; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wctype()* function determines a classification rule for wide-character codes according to the category `LC_CTYPE`, particularly for use with [iswctype\(\)](#) (p. 1463).

Some classes are defined in all locales, although additional classes may be defined for `LC_CTYPE`. Use [setlocale\(\)](#) (p. 2880) to modify the category `LC_CTYPE`.

Defined Classes:

- `alnum`
- `alpha`
- `blank`
- `cntrl`
- `digit`
- `graph`
- `lower`
- `print`
- `punct`
- `space`
- `upper`
- `xdigit`

-

Returns:

A `wctype_t` object that you can use in a call to [iswctype\(\)](#) (p. 1463), or 0 if the character class name isn't valid for the current locale.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wmemchr()

Locate the first occurrence of a wide character in a buffer

Synopsis:

```
#include <wchar.h>

wchar_t * wmemchr( const wchar_t * ws,
                  wchar_t wc,
                  size_t n );
```

Arguments:

ws

The buffer that you want to search.

wc

The character that you're looking for.

n

The number of wide characters to search in the buffer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wmemchr()* function locates the first occurrence of *wc* in the first *n* wide characters of the buffer pointed to by *ws*.

The *wmemchr()* function is locale-independent and treats all `wchar_t` values identically, even if they're null or invalid characters.

Returns:

A pointer to the located character, or `NULL` if *wc* couldn't be found.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wmemcmp()

Compare the wide characters in two buffers

Synopsis:

```
#include <wchar.h>

int wmemcmp( const wchar_t * ws1,
             const wchar_t * ws2,
             size_t n );
```

Arguments:

ws1, ws2

The wide-character strings that you want to compare.

n

The number of wide characters to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wmemcmp()* function compares *n* wide characters of the buffer pointed to by *ws1* to those in the buffer pointed to by *ws2*.

Returns:

<0

ws1 is less than *ws2*.

0

ws1 is equal to *ws2*.

>0

ws1 is greater than *ws2*.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wmemcpy()

Copy wide characters from one buffer to another

Synopsis:

```
#include <wchar.h>

wchar_t * wmemcpy( wchar_t * ws1,
                  const wchar_t * ws2,
                  size_t n );
```

Arguments:

ws1

A pointer to the buffer that you want to copy the wide characters into.

ws2

A pointer to the buffer that you want to copy the wide characters from.

n

The number of wide characters to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *wmemcpy()* function copies *n* wide characters from the buffer pointed to by *ws2* into the buffer pointed to by *ws1*.

The *wmemcpy()* function is locale-independent and treats all `wchar_t` values identically, even if they're null or invalid characters.



Copying overlapping buffers isn't guaranteed to work; use [wmemmove\(\)](#) (p. 3730) to copy buffers that overlap.

Returns:

A pointer to the destination buffer (i.e., the same pointer as *ws1*).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wmemmove()

Copy wide characters from one buffer to another

Synopsis:

```
#include <wchar.h>

wchar_t * wmemmove( wchar_t * ws1,
                    const wchar_t * ws2,
                    size_t n );
```

Arguments:

ws1

A pointer to where you want the function to copy the data.

ws2

A pointer to the buffer that you want to copy data from.

n

The number of wide characters to copy.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memmove()` function copies `n` wide characters from the buffer pointed to by `ws2` to the buffer pointed to by `ws1`. This function copies overlapping regions safely.

The `wmemmove()` function is locale-independent and treats all `wchar_t` values identically, even if they're null or invalid characters.



Use [wmemcpy\(\)](#) (p. 3728) for greater speed when copying buffers that don't overlap.

Returns:

A pointer to the destination buffer (i.e. the same pointed as `ws1`).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wmemset()

Set wide characters in memory

Synopsis:

```
#include <wchar.h>

wchar_t * wmemset( wchar_t * ws,
                  wchar_t wc,
                  size_t n );
```

Arguments:

ws

A pointer to the memory that you want to set.

wc

The value that you want to store in each wide character.

length

The number of wide characters to set.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *memset()* function fills *n* wide characters starting at *ws* with the value *wc*.

The *wmemset()* function is locale-independent and treats all *wchar_t* values identically, even if they're null or invalid wide characters.

Returns:

A pointer to the destination buffer (i.e. the same pointer as *ws*).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

wordexp()

Perform word expansions

Synopsis:

```
#include <wordexp.h>

int wordexp( const char * words,
             wordexp_t * pwordexp,
             int flags );
```

Library:

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The C bindings for performing word expansions aren't currently supported.

Returns:

`WRDE_NOSYS` to indicate that *wordexp()* isn't currently supported.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wordfree()

Free a word expansion buffer

Synopsis:

```
#include <wordexp.h>

void wordfree( wordexp_t * pwordexp );
```

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.



This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The C bindings for performing word expansions aren't currently supported.

Classification:

[POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wprintf()

Write formatted output to stdout

Synopsis:

```
#include <wchar.h>

int wprintf( const wchar_t* format,
            ... );
```

Arguments:

format

A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see [printf\(\)](#) (p. 2133).

Library:

libc

Use the -l c option to gcc to link against this library. This library is usually included automatically.

Description:

The *wprintf()* function writes output to the *stdout* stream, under control of the argument *format*. It's the wide-character version of [printf\(\)](#) (p. 2133).

Returns:

The number of characters written, or a negative value if an output error occurred ([errno](#) (p. 539) is set).

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

write()

Write bytes to a file

Synopsis:

```
#include <unistd.h>

ssize_t write( int fildes,
               const void* buf,
               size_t nbytes );
```

Arguments:

fildes

The file descriptor for the file you want to write in.

buf

A pointer to a buffer that contains the data you want to write.

nbytes

The number of bytes to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The *write()* function attempts to write *nbytes* bytes to the file associated with the open file descriptor, *fildes*, from the buffer pointed to by *buf*.

If *nbytes* is zero, *write()* returns zero, and has no other effect.

On a regular file or other file capable of seeking, and if `O_APPEND` isn't set, *write()* starts at a position in the file given by the file offset associated with *fildes*. If `O_APPEND` is set, the file offset is set to the end of file before each write operation. Before successfully returning from *write()*, the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file is set to this file offset.



Note that the *write()* call ignores advisory locks that may have been set by the *fcntl()* (p. 623) function.

On a file not capable of seeking, *write()* starts at the current position.

If *write()* requests that more bytes be written than there's room for (for example, all blocks on a disk are already allocated), only as many bytes as there's room for are written. For example, if there's only room for 80 more bytes in a file, a write of 512 bytes would return 80. The next write of a nonzero number of bytes would give a failure return (except as noted below).

When *write()* returns successfully, its return value is the number of bytes actually written to the file. This number is never greater than *nbytes*, although it may be less than *nbytes* under certain circumstances detailed below.

If *write()* is interrupted by a signal before it has written any data, it returns a value of -1, and *errno* (p. 539) is set to *EINTR*. However, if *write()* is interrupted by a signal after it has successfully written some data, it returns the number of bytes written.

If the value of *nbytes* is greater than *INT_MAX*, *write()* returns -1 and sets *errno* to *EINVAL*. See `<limits.h>`.

Write requests to a pipe (or FIFO) are handled the same as a regular file, with the following exceptions:

- There's no file offset associated with a pipe, therefore each write request appends to the end of the pipe.
- Write requests of *PIPE_BUF* bytes or less aren't interleaved with data from other processes doing writes on the same pipe. Writes of greater than *PIPE_BUF* bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the *O_NONBLOCK* flag is set.
- If the *O_NONBLOCK* flag is clear, a write request may cause the process to block, but on normal completion it returns *nbytes*.
- If the *O_NONBLOCK* flag is set, write requests are handled differently, in the following ways:
 - The *write()* function doesn't block the process.
 - Write requests for *PIPE_BUF* bytes or less either succeed completely and return *nbytes*, or return -1 and *errno* is set to *EAGAIN*.

If you call *write()* with *nbytes* greater than *PIPE_BUF* bytes, it either transfers what it can and returns the number of bytes written, or transfers no data, returning -1 and setting *errno* to *EAGAIN*. Also, if *nbytes* is greater than *PIPE_BUF* bytes and all data previously written to the pipe has been read (that is, the pipe is empty), *write()* transfers at least *PIPE_BUF* bytes.

When attempting to write to a file (other than a pipe or FIFO) that supports nonblocking writes and can't accept the data immediately:

- If the `O_NONBLOCK` flag is clear, `write()` blocks until the data can be accepted.
- If the `O_NONBLOCK` flag is set, `write()` doesn't block the process. If some data can be written without blocking the process, `write()` transfers what it can and returns the number of bytes written. Otherwise, it returns -1 and sets `errno` to `EAGAIN`.

If `write()` is called with the file offset beyond the end-of-file, the file is extended to the current file offset with the intervening bytes filled with zeroes. This is a useful technique for pregrowing a file.

If `write()` succeeds, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Returns:

The number of bytes written, or -1 if an error occurred (`errno` (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor, *fdes*, isn't a valid file descriptor open for writing.

ECONNRESET

A write was attempted on a socket that isn't connected.

EFBIG

One of the following occurred:

- An attempt was made to write a file that exceeds the maximum file size or the process's file size limit, and there was no room for any bytes to be written.
- The file is a regular file, *nbytes* is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fdes*.

EINTR

The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EIO

One of the following:

- The process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU**, and the process group of the process is orphaned.
- A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.
- The filesystem resides on a removable media device, and the media has been forcibly removed.

ENETDOWN

A write was attempted on a socket and the local network interface used to reach the destination is down.

ENETUNREACH

A write was attempted on a socket and no route to the network is present.

ENOSPC

There's no free space remaining on the device containing the file.

ENOSYS

The *write()* function isn't implemented for the filesystem specified by *filedes*.

ENXIO

One of the following occurred:

- A request was made of a nonexistent device, or the request was outside the capabilities of the device.
- A hangup occurred on the **STREAM** being written to.

EPIPE

One of the following occurred:

- An attempt was made to write to a pipe (or FIFO) that isn't open for reading by any process, or that has only one end open. A **SIGPIPE** signal is also sent to the process.
- A write was attempted on a socket that is shut down for writing, or is no longer connected. In the latter case, if the socket is of type **SOCK_STREAM**, a **SIGPIPE** signal is delivered to the calling process.

ERANGE

The transfer request size was outside the range supported by the STREAMS file associated with *fildev*.

Examples:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

char buffer[] = { "A text record to be written" };

int main( void )
{
    int  fd;
    int  size_written;

    /* open a file for output */
    /* replace existing file if it exists */
    fd = creat( "myfile.dat", S_IRUSR | S_IWUSR );

    /* write the text */
    size_written = write( fd, buffer,
        sizeof( buffer ) );

    /* test for error */
    if( size_written != sizeof( buffer ) ) {
        perror( "Error writing myfile.dat" );
        return EXIT_FAILURE;
    }

    /* close the file */
    close( fd );

    return EXIT_SUCCESS;
}
```

Classification:

[POSIX 1003.1 XSI](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

writeblock()

Write blocks of data to a file

Synopsis:

```
#include <unistd.h>

int writeblock( int fd,
                size_t blksize,
                unsigned block,
                int numblks,
                const void *buff );
```

Arguments:

fd

The file descriptor for the file you want to write in.

blksize

The number of bytes in each block of data.

block

The block number from which to start writing. Blocks are numbered starting at 0.

numblks

The number of blocks to write.

buff

A pointer to a buffer that contains the blocks of data that you want to write.

Library:

libc

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

Description:

The *writeblock()* function writes *numblks* blocks of data to the file associated with the open file descriptor, *fd*, from the buffer pointed to by *buff*, starting at block number *block*.

This function is useful for direct updating of raw blocks on a block special device (for example, raw disk blocks), but you can also use it for high-speed updating (for example, of database files). The speed gain is through the combined seek/write implicit in this call.

If *numblks* is zero, *writeblock()* returns zero, and has no other results.

If successful, *writeblock()* returns the number of blocks actually written to the disk associated with *fd*. This number is never greater than *numblks*, but could be less than *numblks* if one of the following occurs:

- The process attempts to write more blocks than implementation limits allow to be written in a single atomic operation.
- A write error occurred after writing at least one block, and you set one of the sync flags (`O_SYNC` or `O_DSYNC` — see [open\(\)](#) (p. 1905)) when you opened the file.

If a write error occurs on the first block and one of the sync flags is set, *writeblock()* returns -1 and sets [errno](#) (p. 539) to `EIO`.

If one of the sync flags is set, *writeblock()* doesn't return until the blocks are actually transferred to the disk. If neither of the flags is set, *writeblock()* places the blocks in the cache and schedules them for writing as soon as possible, but returns before the writing takes place.



In the latter instance, it's impossible for the application to know if the write succeeded or not (due to system failures or bad disk blocks). Using the sync flags significantly impacts the performance of *writeblock()*, but guarantees that the data can be recovered.

Returns:

The number of blocks actually written. If an error occurred, *writeblock()* returns -1, sets [errno](#) (p. 539) to indicate the error, and doesn't change the contents of the buffer pointed to by *buff*.

Errors:

EBADF

The *fd* argument isn't a valid file descriptor that's open for writing a block-oriented device.

EIO

A physical write error occurred on the first block, and either `O_DSYNC` or `O_SYNC` is set.

EINVAL

The starting position is invalid (0 or negative), or beyond the end of the file.

Classification:

QNX Neutrino (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

writev()

Write bytes to a file

Synopsis:

```
#include <sys/uio.h>

ssize_t writev( int fildev,
                const iove_t* iov,
                int iovcnt );
```

Arguments:

fildev

The file descriptor for the file you want to write in.

iov

An array of `iovec_t` objects that contain the data that you want to write.

iovcnt

The number of elements in the array.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `writev()` function performs the same action as [write\(\)](#) (p. 3737), but gathers the output data from the `iovcnt` buffers specified by the members of the `iov` array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt-1]`.

For `writev()`, the `iovec_t` structure contains the following members:

iov_base

Base address of a memory area from which data should be written.

iov_len

The length of the memory area.

The `writev()` function always writes a complete area before proceeding to the next.

The maximum number of entries in the *iov* array is `UIO_MAXIOV`.



Note that *writev()* ignores advisory locks that may have been set by the *fcntl()* (p. 623) function.

If *writev()* is interrupted by a signal before it has written any data, it returns a value of -1, and *errno* is set to `EINTR`. However, if *writev()* is interrupted by a signal after it has successfully written some data, it will return the number of bytes written.

For more details, see the *write()* (p. 3737) function.

Returns:

The number of bytes written, or -1 if an error occurs (*errno* (p. 539) is set).

Errors:

EAGAIN

The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF

The file descriptor, *filides*, isn't a valid file descriptor open for writing.

EFBIG

The file is a regular file, where *nbytes* is greater than 0, and the starting position is greater than or equal to the offset maximum associated with the file.

EINTR

The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

EINVAL

The *iovcnt* argument is less than or equal to 0, or greater than `UIO_MAXIOV`.

EIO

A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSPC

There is no free space remaining on the device containing the file.

ENOSYS

The *write()* function isn't implemented for the filesystem specified by *files*.

EPIPE

An attempt was made to write to a pipe (or FIFO) that isn't open for reading by any process. A SIGPIPE signal is also sent to the process.

Classification:

POSIX 1003.1 XSI (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

wscanf()

Scan formatted wide-character input from stdin

Synopsis:

```
#include <wchar.h>

int wscanf( const wchar_t * format,
            ... );
```

Arguments:

format

A wide-character string that specifies the format of the input. For more information, see [scanf\(\)](#) (p. 2713). The formatting string determines what additional arguments you need to provide.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wscanf()` function scans input from *stdin* under control of the *format* argument, assigning values to the remaining arguments. It is the wide-character version of [scanf\(\)](#) (p. 2713) and uses the same conversions.



The `wscanf()` function is generally considered unsafe for string handling; it's safer to use [fgetws\(\)](#) (p. 658) to get a line of input and then use [swscanf\(\)](#) (p. 3274) to process the input.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.

Classification:

[ANSI](#) (p. 46), [POSIX 1003.1](#) (p. 46)

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 26

X

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

There are currently no entries whose names begin with “X”.

Chapter 27

Y

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

This chapter describes the API components whose names begin with “Y”.

y0(), y0f()

Compute a Bessel function of the second kind

Synopsis:

```
#include <math.h>

double y0( double x );
float y0f( float x );
```

Arguments:

x

The number that you want to compute the Bessel function for.

Library:

libbessel

Use the -l bessel option to gcc to link against this library.

Description:

Compute the Bessel function of the second kind for *x*.

Returns:

The result of the Bessel function of *x*.

 If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

y0() is [POSIX 1003.1 XSI](#) (p. 46); *y0f()* is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

y1(), y1f()

Compute a Bessel function of the second kind

Synopsis:

```
#include <math.h>

double y1( double x );
float y1f( float x );
```

Arguments:

x

The number that you want to compute the Bessel function for.

Library:

libbessel

Use the -l bessel option to qcc to link against this library.

Description:

Compute the Bessel function of the second kind for *x*.

Returns:

The result of the Bessel function of *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Examples:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;

    x = j0( 2.4 );
    y = y1( 1.58 );
    z = jn( 3, 2.4 );

    printf( "j0(2.4) = %f, y1(1.58) = %f\n", x, y );
```



```
    printf( "jn(3,2.4) = %f\n", z );  
    return EXIT_SUCCESS;  
}
```

Classification:

y1() is [POSIX 1003.1 XSI](#) (p. 46); y1f() is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

yn(), ynf()

Compute a Bessel function of the second kind

Synopsis:

```
#include <math.h>

double yn( int n,
           double x );

float ynf( int n,
           float x );
```

Arguments:

n, x

The numbers that you want to compute the Bessel function for.

Library:

libbessel

Use the -l bessel option to qcc to link against this library.

Description:

Compute the Bessel function of the second kind for *n* and *x*.

Returns:

The result of the Bessel function of *n* and *x*.



If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set [errno](#) (p. 539) to 0, call the function, and then check *errno* again. These functions don't change *errno* if no errors occurred.

Classification:

yn() is [POSIX 1003.1 XSI](#) (p. 46); *ynf()* is [Unix](#) (p. 46)

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

Chapter 28

Z

The functions, macros, and data structures in the C library are described here in alphabetical order.

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

There are currently no entries whose names begin with “Z”.

Appendix A

SOCKS — A Basic Firewall

This appendix describes SOCKS, a basic firewall.

About SOCKS

SOCKS is a package consisting of a proxy server, client programs, and a library (`libsocks`) for adapting other applications into new client programs.

The original SOCKS was written by David Koblas (`koblas@netcom.com`). The SOCKS protocol has changed over time. The client library shipped as of printing corresponds to SOCKS v4.2. Since the server and the clients must use the same SOCKS protocol, this library doesn't work with servers of previous releases; clients compiled with these libraries won't work with older servers.

How to SOCKSify a client



If your client is using [UDP](#) (p. 3553) to transfer data, you can't use SOCKS. To see if your client uses UDP, search for the string "SOCK_DGRAM" in your source.

1. At or near the beginning of [main\(\)](#) (p. 1566), you can add a call to [SOCKSinit\(\)](#) (p. 3081).

You can omit this step; the only reason for calling [SOCKSinit\(\)](#) directly is to associate a name with your SOCKS client (rather than the generic "SOCKSclient" default string).

2. Add the following options to your compile commands:

```
-Dconnect=Rconnect -Dgetsockname=Rgetsockname \
-Dbind=Rbind -Daccept=Raccept -Dlisten=Rlisten \
-Drcmd=Rrcmd -Dselect=Rselect
```

If you're using a `Makefile`, add these options to the definition of macro `CFLAGS`.

These options replace calls to certain functions with versions that use the SOCKS server:

Non-SOCKS function:	SOCKS function:
accept() (p. 58)	Raccept() (p. 2480)
bind() (p. 208)	Rbind() (p. 2489)
connect() (p. 348)	Rconnect() (p. 2494)
getsockname() (p. 960)	Rgetsockname() (p. 2650)
listen() (p. 1523)	Rlisten() (p. 2656)
rcmd() (p. 2491)	Rrcmd() (p. 2668)
select() (p. 2791)	Rselect() (p. 2672)

3. Link against the SOCKS library by adding `-l socks` to your link line.

If you're using a `Makefile`, simply add this information to the definition of the macro `LDFLAGS`.

For most programs, the above steps should be sufficient to SOCKSify the package. If the above doesn't work, you may need to look at things a little more closely. The next section describes how the SOCKS library expects to be used.

What SOCKS expects

The SOCKS library covers only some of the socket functions, which must be called in a particular order:



You must use [TCP](#) (p. 3359); SOCKS doesn't support UDP.

1. The first socket function invoked must be either [connect\(\)](#) (p. 348) or [rcmd\(\)](#) (p. 2491).
 2. If you call [connect\(\)](#) on a nonblocking socket, no I/O can occur on that socket until another [connect\(\)](#), with the same arguments, returns -1 and sets [errno](#) (p. 539) to EISCONN. This is required even if you use [select\(\)](#) (p. 2791) on write to check the readiness of that socket.
-



While a connection is still pending, don't try to start another connection via [connect\(\)](#), or start a sequence of [bind\(\)](#) (p. 208), [getsockname\(\)](#) (p. 960), [listen\(\)](#) (p. 1523), and [accept\(\)](#) (p. 58).

3. You must call [bind\(\)](#) after a successful [connect\(\)](#) call to a host for a specific service.
4. You must follow the call to [bind\(\)](#) by calls to [getsockname\(\)](#), [listen\(\)](#), and [accept\(\)](#), in that order.

Most client programs fit these assumptions very well and can be SOCKSified without changing the code at all using the steps described in “[How to SOCKSify a client.](#)”

Some client programs use a [bind\(\)](#) before each [connect\(\)](#). If the [bind\(\)](#) is used to claim a specific port or a specific network interface, the current SOCKS library can't accommodate such use. Very often though, such a [bind\(\)](#) call is there for no specific reason and may simply be deleted.

Appendix B

Full Safety Information

This appendix includes the full safety information for all the functions described in this reference.

A

Function	Classification	Cancel	ISR	Signal	Thread
abort() (p. 54)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
abs() (p. 56)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
accept() (p. 58)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
access() (p. 61)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
acl_add_perm() (p. 64)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_calc_mask() (p. 66)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_clear_perms() (p. 68)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_copy_entry() (p. 70)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_copy_ext() (p. 72)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_copy_int() (p. 74)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_create_entry() (p. 76)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_delete_entry() (p. 78)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_delete_perm() (p. 80)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_dup() (p. 82)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_free() (p. 84)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_from_text() (p. 86)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_get_entry() (p. 88)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_get_fd() (p. 90)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_get_file() (p. 92)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_get_permset() (p. 94)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_get_qualifier() (p. 96)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_get_tag_type() (p. 98)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_init() (p. 100)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_set_fd() (p. 102)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_set_file() (p. 104)	QNX Neutrino (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
acl_set_permset() (p. 106)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_set_qualifier() (p. 108)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_set_tag_type() (p. 110)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_size() (p. 112)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_to_text() (p. 114)	QNX Neutrino (p. 46)	No	No	No	Yes
acl_valid() (p. 116)	QNX Neutrino (p. 46)	No	No	No	Yes
acos() (p. 118)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
acosf() (p. 118)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
acosl() (p. 118)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
acosh() (p. 120)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
acoshf() (p. 120)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
acoshl() (p. 120)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
aio_cancel() (p. 124)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
aio_error() (p. 127)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
aio_fsync() (p. 129)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
aio_read() (p. 131)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
aio_read64() (p. 131)	Large-file support (p. 46)	No	No	Yes	Yes
aio_return() (p. 134)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
aio_suspend() (p. 136)	POSIX 1003.1 AIO (p. 46)	Yes	No	Yes	Yes
aio_write() (p. 138)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
aio_write64() (p. 138)	Large-file support (p. 46)	No	No	Yes	Yes
alarm() (p. 144)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
alloca() (p. 147)	Unix (p. 46)	No	No	Yes	Yes
alphasort() (p. 149)	Legacy Unix (p. 46)	No	Yes	Yes	Yes
asctime() (p. 155)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	No
asctime_r() (p. 155)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
asin() (p. 157)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
asinf() (p. 157)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
asinl() (p. 157)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
asinh() (p. 159)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
asinhf() (p. 159)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
asinhll() (p. 159)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
assert() (p. 161)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atan() (p. 163)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atanf() (p. 163)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atanll() (p. 163)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atan2() (p. 165)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atan2f() (p. 165)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atanh() (p. 167)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atanhf() (p. 167)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atanhll() (p. 167)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atexit() (p. 169)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atof() (p. 171)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
atoh() (p. 173)	QNX 4 (p. 46)	No	Yes	Yes	Yes
atoi() (p. 175)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
atol() (p. 177)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
atoll() (p. 177)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
atomic_add() (p. 179)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_add_value() (p. 181)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_clr() (p. 183)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_clr_value() (p. 185)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_set() (p. 187)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_set_value() (p. 189)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_sub() (p. 191)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_sub_value() (p. 193)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_toggle() (p. 195)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
atomic_toggle_value() (p. 197)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes

B

Function	Classification	Cancel	ISR	Signal	Thread
<i>basename()</i> (p. 202)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	Yes	Yes	Yes
<i>bcmp()</i> (p. 204)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	Yes	Yes	Yes
<i>bcopy()</i> (p. 206)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	Yes	Yes	Yes
<i>bind()</i> (p. 208)	<i>POSIX 1003.1</i> (p. 46)	No	No	No	Yes
<i>bindresvport()</i> (p. 211)	<i>Unix</i> (p. 46)	No	No	No	No
<i>brk()</i> (p. 213)	<i>Legacy Unix</i> (p. 46)	No	No	No	Yes
<i>bsearch()</i> (p. 216)	<i>ANSI</i> (p. 46) <i>POSIX 1003.1</i> (p. 46)	No	Yes	Yes	Yes
<i>bt_get_backtrace()</i> (p. 219)	<i>QNX Neutrino</i> (p. 46)	No	No	Yes	Yes
<i>bt_init_accessor()</i> (p. 221)	<i>QNX Neutrino</i> (p. 46)	No	No	Yes	Yes
<i>bt_load_memmap()</i> (p. 223)	<i>QNX Neutrino</i> (p. 46)	No	No	No	Yes
<i>bt_release_accessor()</i> (p. 225)	<i>QNX Neutrino</i> (p. 46)	No	Yes	Yes	Yes
<i>bt_set_flags()</i> (p. 227)	<i>QNX Neutrino</i> (p. 46)	No	Yes	Yes	Yes
<i>bt_sprn_memmap()</i> (p. 229)	<i>QNX Neutrino</i> (p. 46)	No	No	Yes	Yes
<i>bt_sprnf_addrs()</i> (p. 231)	<i>QNX Neutrino</i> (p. 46)	No	No	No	Yes
<i>bt_translate_addrs()</i> (p. 234)	<i>QNX Neutrino</i> (p. 46)	No	No	Yes	Yes
<i>bt_unload_memmap()</i> (p. 236)	<i>QNX Neutrino</i> (p. 46)	No	No	No	Yes
<i>btowc()</i> (p. 238)	<i>ANSI</i> (p. 46) <i>POSIX 1003.1</i> (p. 46)	No	No	Yes	Yes
<i>bzero()</i> (p. 239)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	Yes	Yes	Yes

C

Function	Classification	Cancel	ISR	Signal	Thread
cabs() (p. 242)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cabsf() (p. 242)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cache_fini() (p. 244)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
CACHE_FLUSH() (p. 246)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
cache_init() (p. 248)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
CACHE_INVALID() (p. 253)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
calloc() (p. 255)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cbrt() (p. 258)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cbrtf() (p. 258)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ceil() (p. 260)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ceilf() (p. 260)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cfgetispeed() (p. 262)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
cfgetospeed() (p. 264)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
cfgopen() (p. 266)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
cfmakeraw() (p. 270)	Unix (p. 46)	No	No	Yes	Yes
cfree() (p. 272)	Unix (p. 46)	No	No	No	Yes
cfsetispeed() (p. 274)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
cfsetospeed() (p. 276)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
ChannelCreate() (p. 278)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ChannelCreate_r() (p. 278)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ChannelDestroy() (p. 284)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ChannelDestroy_r() (p. 284)	QNX Neutrino (p. 46)	No	No	Yes	Yes
chdir() (p. 286)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
chmod() (p. 289)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
chown() (p. 292)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
chroot() (p. 295)	Legacy Unix (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
chsize() (p. 298)	QNX 4 (p. 46)	Yes	No	Yes	Yes
clearenv() (p. 300)	QNX 4 (p. 46)	No	No	No	Yes
clearerr() (p. 302)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ClientInfoExtFree() (p. 304)	QNX Neutrino (p. 46)	No	No	Yes	Yes
clock() (p. 305)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
clock_getcpuclockid() (p. 307)	POSIX 1003.1 CPT (p. 46)	No	No	Yes	Yes
clock_getres() (p. 309)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
clock_gettime() (p. 312)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
clock_nanosleep() (p. 315)	POSIX 1003.1 CS (p. 46)	Yes	No	Yes	Yes
clock_settime() (p. 319)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
ClockAdjust() (p. 322)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockAdjust_r() (p. 322)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockCycles() (p. 325)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ClockId() (p. 327)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockId_r() (p. 327)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockPeriod() (p. 329)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockPeriod_r() (p. 329)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockTime() (p. 333)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ClockTime_r() (p. 333)	QNX Neutrino (p. 46)	No	No	Yes	Yes
close() (p. 336)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
closedir() (p. 338)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
closelog() (p. 340)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	Yes
confstr() (p. 344)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
connect() (p. 348)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
ConnectAttach() (p. 351)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ConnectAttach_r() (p. 351)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ConnectClientInfo() (p. 356)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ConnectClientInfo_r() (p. 356)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ConnectClientInfoAble() (p. 360)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
ConnectClientInfoExt() (p. 363)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ConnectDetach() (p. 367)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ConnectDetach_r() (p. 367)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ConnectFlags() (p. 369)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ConnectFlags_r() (p. 369)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ConnectServerInfo() (p. 372)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ConnectServerInfo_r() (p. 372)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
copysign() (p. 375)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
copysignf() (p. 375)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cos() (p. 377)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cosf() (p. 377)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cosl() (p. 377)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
cosh() (p. 379)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
coshf() (p. 379)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
coshl() (p. 379)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
creat() (p. 381)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
creat64() (p. 381)	Large-file support (p. 46)	Yes	No	Yes	Yes
crypt() (p. 385)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
ctermid() (p. 387)	POSIX 1003.1 (p. 46)	No	No	No	Read the Caveats
ctime() (p. 389)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	No
ctime_r() (p. 389)		No	No	Yes	Yes

D

Function	Classification	Cancel	ISR	Signal	Thread
daemon() (p. 394)	Legacy Unix (p. 46)	No	No	Yes	No
DebugBreak() (p. 397)	QNX Neutrino (p. 46)	No	No	Yes	Yes
DebugKDBreak() (p. 398)	QNX Neutrino (p. 46)	No	No	Yes	Yes
DebugKDOOutput() (p. 399)	QNX Neutrino (p. 46)	No	No	Yes	Yes
delay() (p. 401)	QNX 4 (p. 46)	Yes	No	Yes	Yes
devctl() (p. 403)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
dhcpcctl*() (p. 412)	NetBSD (p. 46)		No	No	No
difftime() (p. 417)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
dircntl() (p. 419)	QNX Neutrino (p. 46)	No	No	No	Yes
dirname() (p. 424)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
dispatch_block() (p. 426)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
dispatch_context_alloc() (p. 429)	QNX Neutrino (p. 46)	No	No	No	Yes
dispatch_context_free() (p. 431)	QNX Neutrino (p. 46)	No	No	No	Yes
dispatch_create() (p. 433)	QNX Neutrino (p. 46)	No	No	No	Yes
dispatch_create_channel() (p. 436)	QNX Neutrino (p. 46)	No	No	No	Yes
dispatch_destroy() (p. 438)	QNX Neutrino (p. 46)	No	No	No	Yes
dispatch_handler() (p. 440)	QNX Neutrino (p. 46)	Read the Caveats	No	No	Yes
dispatch_timeout() (p. 443)	QNX Neutrino (p. 46)	No	No	No	Yes
dispatch_unblock() (p. 445)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
div() (p. 447)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
dladdr() (p. 449)	Unix (p. 46)	No	No	No	Yes
dlclose() (p. 451)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	Yes
dlopen() (p. 453)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
dlopen() (p. 454)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	Yes
dlsym() (p. 461)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
dn_comp() (p. 464)	Unix (p. 46)	No	No	Yes	Yes
dn_expand() (p. 466)	Unix (p. 46)	No	No	Yes	Yes
drand48() (p. 468)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
ds_clear() (p. 469)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ds_create() (p. 471)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ds_deregister() (p. 474)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ds_flags() (p. 476)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ds_get() (p. 478)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ds_register() (p. 480)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ds_set() (p. 481)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
dup() (p. 483)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
dup2() (p. 485)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes

E

Function	Classification	Cancel	ISR	Signal	Thread
eaccess() (p. 488)	Unix (p. 46)	No	No	Yes	Yes
encrypt() (p. 492)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
endsent() (p. 495)	NetBSD (p. 46)	Yes	No	No	No
endgrent() (p. 496)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
endhostent() (p. 497)	POSIX 1003.1 (p. 46)	Yes	No	No	No
ENDIAN_BE16() (p. 498)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_BE32() (p. 500)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_BE64() (p. 502)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_LE16() (p. 504)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_LE32() (p. 506)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_LE64() (p. 508)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_RET16() (p. 510)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_RET32() (p. 512)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_RET64() (p. 514)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_SWAP16() (p. 516)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_SWAP32() (p. 518)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
ENDIAN_SWAP64() (p. 520)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
endnetent() (p. 522)	POSIX 1003.1 (p. 46)	Yes	No	No	No
endprotoent() (p. 523)	POSIX 1003.1 (p. 46)	Yes	No	No	No
endpwent() (p. 524)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
endservent() (p. 525)	POSIX 1003.1 (p. 46)	Yes	No	No	No
endspent() (p. 526)	Unix (p. 46)	Yes	No	No	No
endutent() (p. 527)	Unix (p. 46)	Yes	No	No	No
eof() (p. 529)	QNX 4 (p. 46)	Yes	No	Yes	Yes
erand48() (p. 531)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
erf() (p. 532)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
erff() (p. 532)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
erfc() (p. 534)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
erfcf() (p. 534)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
erfcl() (p. 534)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
err() (p. 536)	Unix (p. 46)	Yes	No	Yes	Yes
errx() (p. 536)	Unix (p. 46)	Yes	No	Yes	Yes
execl() (p. 550)	POSIX 1003.1 (p. 46)	No	No	No	Yes
execle() (p. 556)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
execvp() (p. 562)	POSIX 1003.1 (p. 46)	No	No	No	Yes
execvpe() (p. 568)	QNX 4 (p. 46)	No	No	No	Yes
execv() (p. 573)	POSIX 1003.1 (p. 46)	No	No	No	Yes
execve() (p. 579)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
execvp() (p. 585)	POSIX 1003.1 (p. 46)	No	No	No	Yes
execvpe() (p. 591)	QNX 4 (p. 46)	No	No	Yes	Yes
exit() (p. 599)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
exp() (p. 601)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
expf() (p. 601)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
expl() (p. 601)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
expm1() (p. 603)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
expm1f() (p. 603)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes

F

Function	Classification	Cancel	ISR	Signal	Thread
fabs() (p. 606)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fabsf() (p. 606)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fcgopen() (p. 608)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
fchdir() (p. 610)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
fchmod() (p. 612)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
fchown() (p. 615)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
fclose() (p. 618)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fcloseall() (p. 621)	QNX 4 (p. 46)	Yes	No	No	Yes
fcntl() (p. 623)	POSIX 1003.1 (p. 46)	Read the Caveats	No	Yes	Yes
fdatasync() (p. 631)	POSIX 1003.1 SIO (p. 46)	Yes	No	Yes	Yes
fdistrusted() (p. 633)	QNX Neutrino (p. 46)	No	No	Yes	Yes
fdopen() (p. 635)	POSIX 1003.1 (p. 46)	No	No	No	Yes
feof() (p. 637)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ferror() (p. 639)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fflush() (p. 641)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
ffs() (p. 644)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
fgetc() (p. 645)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fgetchar() (p. 648)	QNX 4 (p. 46)	Yes	No	No	Yes
fgetpos() (p. 650)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fgets() (p. 652)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fgetspent() (p. 654)	Unix (p. 46)	Yes	No	No	No
fgetwc() (p. 656)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fgetws() (p. 658)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fileno() (p. 661)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
flink() (p. 663)	Unix (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
flock() (p. 666)	Unix (p. 46)	No	No	Yes	Yes
flockfile() (p. 669)	POSIX 1003.1 TSF (p. 46)	No	No	No	Yes
floor() (p. 671)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
floorf() (p. 671)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
flushall() (p. 673)	QNX 4 (p. 46)	Yes	No	No	Yes
fmod() (p. 675)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fmodf() (p. 675)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fmodl() (p. 675)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fnmatch() (p. 677)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
fopen() (p. 681)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fopen64() (p. 681)	Large-file support (p. 46)	Yes	No	No	Yes
fork() (p. 685)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
forkpty() (p. 688)	Unix (p. 46)	Yes	No	Yes	Yes
fp_exception_mask() (p. 690)	QNX Neutrino (p. 46)	No	No	No	Yes
fp_exception_value() (p. 693)	QNX Neutrino (p. 46)	No	No	No	Yes
fp_precision() (p. 695)	QNX Neutrino (p. 46)	No	No	No	Yes
fp_rounding() (p. 697)	QNX Neutrino (p. 46)	No	No	No	Yes
fpathconf() (p. 699)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
fprintf() (p. 702)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fputc() (p. 704)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fputchar() (p. 707)	QNX 4 (p. 46)	Yes	No	No	Yes
fputs() (p. 709)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fputwc() (p. 711)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fputws() (p. 713)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fread() (p. 715)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
free() (p. 717)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
freeaddrinfo() (p. 719)	POSIX 1003.1 (p. 46)	No	No	No	Yes
freeifaddrs() (p. 720)	Unix (p. 46)	No	No	No	Yes
freopen() (p. 721)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
freopen64() (p. 721)	Large-file support (p. 46)	Yes	No	No	Yes
frexp() (p. 725)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
frexpf() (p. 725)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
fs_crypto_check() (p. 727)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_add() (p. 729)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_key_change() (p. 732)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_key_check() (p. 735)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_key_size() (p. 738)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_lock() (p. 739)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_query() (p. 741)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_remove() (p. 743)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_domain_unlock() (p. 745)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_enable() (p. 748)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_enable_option() (p. 750)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_file_get_domain() (p. 753)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_key_gen() (p. 755)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_migrate_control() (p. 757)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_migrate_status() (p. 760)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_migrate_path() (p. 762)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_migrate_tag() (p. 764)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_file_set_domain() (p. 766)	QNX Neutrino (p. 46)	No	No	No	Yes
fs_crypto_set_logging() (p. 769)	QNX Neutrino (p. 46)	No	No	No	Yes
fscanf() (p. 770)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
fseek() (p. 772)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
fseeko() (p. 772)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
fseeko64() (p. 772)	Large-file support (p. 46)	Yes	No	Yes	Yes
fsetpos() (p. 776)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
fstat() (p. 779)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
fstat64() (p. 779)	Large-file support (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
<i>fstatvfs()</i> (p. 783)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
<i>fstatvfs64()</i> (p. 783)	Large-file support (p. 46)	No	No	Yes	Yes
<i>fsync()</i> (p. 787)	POSIX 1003.1 FSC (p. 46)	Yes	No	Yes	Yes
<i>ftell()</i> (p. 789)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
<i>ftello()</i> (p. 789)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
<i>ftello64()</i> (p. 789)	Large-file support (p. 46)	Yes	No	No	Yes
<i>ftime()</i> (p. 791)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
<i>ftruncate()</i> (p. 793)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
<i>ftruncate64()</i> (p. 793)	Large-file support (p. 46)	No	No	Yes	Yes
<i>ftrylockfile()</i> (p. 796)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
<i>ftw()</i> (p. 798)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
<i>ftw64()</i> (p. 798)	Large-file support (p. 46)	Yes	No	Yes	Yes
<i>funlockfile()</i> (p. 801)	POSIX 1003.1 TSF (p. 46)	No	No	No	Yes
<i>futime()</i> (p. 802)	Unix (p. 46)	No	No	Yes	Yes
<i>fwide()</i> (p. 804)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
<i>fwprintf()</i> (p. 806)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
<i>fwrite()</i> (p. 808)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
<i>fwscanf()</i> (p. 810)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes

G

Function	Classification	Cancel	ISR	Signal	Thread
<i>gai_strerror()</i> (p. 814)	<i>POSIX 1003.1</i> (p. 46)	No	Yes	Yes	Yes
<i>gamma()</i> (p. 816)	<i>Legacy Unix</i> (p. 46)	No	No	No	Yes
<i>gamma_r()</i> (p. 816)	<i>Legacy Unix</i> (p. 46)	No	No	No	Yes
<i>gammaf()</i> (p. 816)	<i>Legacy Unix</i> (p. 46)	No	No	No	Yes
<i>gammaf_r()</i> (p. 816)	<i>Legacy Unix</i> (p. 46)	No	No	No	Yes
<i>getaddrinfo()</i> (p. 818)	<i>POSIX 1003.1</i> (p. 46)	Yes	No	No	Yes
<i>getc()</i> (p. 823)	<i>ANSI</i> (p. 46) <i>POSIX 1003.1</i> (p. 46)	Yes	No	No	Yes
<i>getc_unlocked()</i> (p. 825)	<i>POSIX 1003.1 TSF</i> (p. 46)	Yes	No	No	No
<i>getchar()</i> (p. 827)	<i>ANSI</i> (p. 46) <i>POSIX 1003.1</i> (p. 46)	Yes	No	No	Yes
<i>getchar_unlocked()</i> (p. 829)	<i>POSIX 1003.1 TSF</i> (p. 46)	Yes	No	No	No
<i>getcwd()</i> (p. 830)	<i>POSIX 1003.1</i> (p. 46)	Yes	No	No	Yes
<i>getdomainname()</i> (p. 833)	<i>Unix</i> (p. 46)	No	No	Yes	Yes
<i>getdtablesize()</i> (p. 835)	<i>Legacy Unix</i> (p. 46)	No	No	Yes	Yes
<i>getegid()</i> (p. 836)	<i>POSIX 1003.1</i> (p. 46)	No	No	Yes	Yes
<i>getenv()</i> (p. 838)	<i>ANSI</i> (p. 46) <i>POSIX 1003.1</i> (p. 46)	No	No	No	No
<i>geteuid()</i> (p. 840)	<i>POSIX 1003.1</i> (p. 46)	No	No	Yes	Yes
<i>getfsfile()</i> (p. 842)	<i>NetBSD</i> (p. 46)	No	No	Yes	Yes
<i>getfsent()</i> (p. 844)	<i>NetBSD</i> (p. 46)	No	No	Yes	Yes
<i>getfsspec()</i> (p. 847)	<i>NetBSD</i> (p. 46)	No	No	Yes	Yes
<i>getgid()</i> (p. 849)	<i>POSIX 1003.1</i> (p. 46)	No	No	Yes	Yes
<i>getgrent()</i> (p. 851)	<i>POSIX 1003.1 XSI</i> (p. 46)	Yes	No	No	No
<i>getgrgid()</i> (p. 853)	<i>POSIX 1003.1</i> (p. 46)	Yes	No	No	No
<i>getgrgid_r()</i> (p. 855)	<i>POSIX 1003.1 TSF</i> (p. 46)	Yes	No	No	Yes
<i>getgrnam()</i> (p. 857)	<i>POSIX 1003.1</i> (p. 46)	Yes	No	No	No
<i>getgrnam_r()</i> (p. 859)	<i>POSIX 1003.1 TSF</i> (p. 46)	Yes	No	No	Yes
<i>getgrouplist()</i> (p. 861)	<i>Unix</i> (p. 46)	Yes	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
getgroups() (p. 864)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
gethostbyaddr() (p. 866)	POSIX 1003.1 OBS (p. 46)	Yes	No	No	No
gethostbyaddr_r() (p. 869)	Unix (p. 46)	Yes	No	No	Yes
gethostbyname() (p. 872)	POSIX 1003.1 OBS (p. 46)	Yes	No	No	No
gethostbyname2() (p. 872)	QNX Neutrino (p. 46)	Yes	No	No	No
gethostbyname_r() (p. 875)	Unix (p. 46)	Yes	No	No	Yes
gethostent() (p. 878)	POSIX 1003.1 (p. 46)	Yes	No	No	No
gethostent_r() (p. 880)	Unix (p. 46)	Yes	No	No	Yes
gethostname() (p. 883)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
getifaddrs() (p. 885)	Unix (p. 46)	Yes	No	No	Yes
GETIOVBASE() (p. 887)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
GETIOVLEN() (p. 888)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
getitimer() (p. 889)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
getlogin() (p. 891)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getlogin_r() (p. 892)	POSIX 1003.1 TSF (p. 46)	Yes	No	No	Yes
getnameinfo() (p. 894)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
getnetbyaddr() (p. 898)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getnetbyname() (p. 900)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getnetent() (p. 902)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getopt() (p. 903)	POSIX 1003.1 (p. 46)	Read the Caveats	No	No	No
getpagesize() (p. 908)	Legacy Unix (p. 46)	No	No	Yes	Yes
getpagesizes() (p. 909)	Unix (p. 46)	No	No	Yes	Yes
getpagesizes64() (p. 909)	QNX Neutrino (p. 46)	No	No	Yes	Yes
getpass() (p. 911)	Legacy Unix (p. 46)	Yes	No	No	No
getpeereid() (p. 913)	NetBSD (p. 46)	No	No	Yes	Yes
getpeername() (p. 915)	POSIX 1003.1 (p. 46)	No	No	No	Yes
getpgid() (p. 917)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
getpgrp() (p. 919)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
getpid() (p. 920)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
getppid() (p. 921)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
getprio() (p. 922)	QNX 4 (p. 46)	No	No	Yes	Yes
getprotobyname() (p. 924)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getprotobynumber() (p. 926)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getprotoent() (p. 928)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getpwent() (p. 930)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
getpwent_r() (p. 932)	POSIX 1003.1 TSF (p. 46)	No	No	No	Yes
getpwnam() (p. 934)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getpwnam_r() (p. 936)	POSIX 1003.1 TSF (p. 46)	Yes	No	No	Yes
getpwuid() (p. 938)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getpwuid_r() (p. 940)	POSIX 1003.1 TSF (p. 46)	Yes	No	No	Yes
getrlimit() (p. 942)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
getrlimit64() (p. 942)	Large-file support (p. 46)	No	No	Yes	Yes
getrusage() (p. 945)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
gets() (p. 950)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
getservbyname() (p. 952)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getservbyport() (p. 954)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getservent() (p. 956)	POSIX 1003.1 (p. 46)	Yes	No	No	No
getsid() (p. 958)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
getsockname() (p. 960)	POSIX 1003.1 (p. 46)	No	No	No	Yes
getsockopt() (p. 962)	POSIX 1003.1 (p. 46)	No	No	No	Yes
getspent() (p. 971)	Unix (p. 46)	Yes	No	No	No
getspent() (p. 971)	Unix (p. 46)	Yes	No	No	Yes
getspent_r() (p. 971)	Unix (p. 46)	Yes	No	No	No
getspent_r() (p. 971)	Unix (p. 46)	Yes	No	No	Yes
getspnam() (p. 974)	Unix (p. 46)	Yes	No	No	No
getspnam() (p. 974)	Unix (p. 46)	Yes	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
getspnam_r() (p. 974)	Unix (p. 46)	Yes	No	No	No
getspnam_r() (p. 974)	Unix (p. 46)	Yes	No	No	Yes
getsubopt() (p. 977)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
gettid() (p. 981)	QNX Neutrino (p. 46)	No	No	Yes	Yes
gettimeofday() (p. 982)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
getuid() (p. 984)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
getutent() (p. 985)	Unix (p. 46)	Yes	No	No	Yes
getutid() (p. 987)	Unix (p. 46)	Yes	No	No	Yes
getutline() (p. 989)	Unix (p. 46)	Yes	No	No	Yes
getw() (p. 991)	Legacy Unix (p. 46)	Yes	No	Yes	Yes
getwc() (p. 993)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
getwchar() (p. 995)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
getwd() (p. 997)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
glob() (p. 999)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
globfree() (p. 1004)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
gmtime() (p. 1005)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	No
gmtime_r() (p. 1007)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
grantpt() (p. 1009)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes

H

Function	Classification	Cancel	ISR	Signal	Thread
hcreate() (p. 1014)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
hdestroy() (p. 1016)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
herror() (p. 1017)	Unix (p. 46)	Yes	No	No	No
hsearch() (p. 1021)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
hstrerror() (p. 1024)	Unix (p. 46)	No	No	Yes	Yes
htonl() (p. 1026)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
htons() (p. 1027)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
hwi_find_item() (p. 1028)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
hwi_find_tag() (p. 1030)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
hwi_next_item() (p. 1032)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
hwi_next_tag() (p. 1035)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
hwi_off2tag() (p. 1037)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
hwi_tag2off() (p. 1039)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
hypot() (p. 1040)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
hypotf() (p. 1040)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes

I

Function	Classification	Cancel	ISR	Signal	Thread
if_freenameindex() (p. 1049)	POSIX 1003.1 (p. 46)	No	No	No	Yes
if_indextoname() (p. 1050)	POSIX 1003.1 (p. 46)	No	No	No	Yes
if_nameindex() (p. 1052)	POSIX 1003.1 (p. 46)	No	No	No	Yes
if_nametoindex() (p. 1054)	POSIX 1003.1 (p. 46)	No	No	No	Yes
ilogb() (p. 1057)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ilogbf() (p. 1057)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
in8() (p. 1059)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
in8s() (p. 1061)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
in16() (p. 1063)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
inbe16() (p. 1063)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
inle16() (p. 1063)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
in16s() (p. 1065)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
in32() (p. 1067)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
inbe32() (p. 1067)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
inle32() (p. 1067)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
in32s() (p. 1069)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
index() (p. 1071)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
inet_addr() (p. 1073)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
inet_aton() (p. 1075)	Unix (p. 46)	No	No	Yes	Yes
inet_lnaof() (p. 1077)	QNX Neutrino (p. 46)	No	No	Yes	Yes
inet_makeaddr() (p. 1078)	QNX Neutrino (p. 46)	No	No	Yes	Yes
inet_net_ntop() (p. 1080)	QNX Neutrino (p. 46)	No	No	No	Yes
inet_netof() (p. 1083)	QNX Neutrino (p. 46)	No	No	Yes	Yes
inet_net_pton() (p. 1085)	QNX Neutrino (p. 46)	No	No	No	Yes
inet_network() (p. 1087)	QNX Neutrino (p. 46)	No	No	Yes	Yes
inet_ntoa() (p. 1089)	POSIX 1003.1 (p. 46)	No	No	No	No

Function	Classification	Cancel	ISR	Signal	Thread
inet_ntoa_r() (p. 1091)	Unix (p. 46)	No	No	No	Yes
inet_ntop() (p. 1093)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
inet_pton() (p. 1096)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
inet6_option_alloc() (p. 1103)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_option_append() (p. 1105)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_option_find() (p. 1107)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_option_init() (p. 1109)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_option_next() (p. 1111)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_option_space() (p. 1113)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_add() (p. 1115)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_getaddr() (p. 1117)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_getflags() (p. 1119)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_init() (p. 1121)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_lasthop() (p. 1123)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_reverse() (p. 1125)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_segments() (p. 1127)	RFC 2292 (p. 46)	No	No	Yes	Yes
inet6_rthdr_space() (p. 1129)	RFC 2292 (p. 46)	No	No	Yes	Yes
initgroups() (p. 1131)	Unix (p. 46)	No	No	No	No
initstate() (p. 1133)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
inotify_add_watch() (p. 1135)	Linux (p. 46)	No	No	No	Yes
inotify_init() (p. 1139)	Linux (p. 46)	No	No	No	Yes
inotify_rm_watch() (p. 1141)	Linux (p. 46)	No	No	No	Yes
input_line() (p. 1143)	QNX 4 (p. 46)	Yes	No	No	No
insque() (p. 1145)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
InterruptAttach() (p. 1147)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptAttach_r() (p. 1147)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptAttachEvent() (p. 1154)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptAttachEvent_r() (p. 1154)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptCharacteristic() (p. 1159)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
InterruptCharacteristic_r() (p. 1159)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptDetach() (p. 1162)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptDetach_r() (p. 1162)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptDisable() (p. 1164)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptEnable() (p. 1166)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptHookIdle() (p. 1168)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptHookTrace() (p. 1171)	QNX Neutrino (p. 46)	No	No	Yes	Yes
InterruptLock() (p. 1174)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptMask() (p. 1176)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptStatus() (p. 1178)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptUnlock() (p. 1179)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptUnmask() (p. 1181)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
InterruptWait() (p. 1183)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
InterruptWait_r() (p. 1183)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ioctl() (p. 1198)	POSIX 1003.1 XSR (p. 46)	Yes	No	No	No
ioctl_socket() (p. 1198)	POSIX 1003.1 XSR (p. 46)	Yes	No	No	No
iofdinfo() (p. 1210)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_acl() (p. 1212)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_acl_default() (p. 1215)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_attr_init() (p. 1217)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_attr_lock() (p. 1219)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
iofunc_attr_trylock() (p. 1226)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
iofunc_attr_unlock() (p. 1228)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_check_access() (p. 1230)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_chmod() (p. 1234)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_chmod_default() (p. 1237)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_chown() (p. 1239)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_chown_default() (p. 1242)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_client_info_able() (p. 1244)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
iofunc_client_info_ext() (p. 1246)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_client_info_ext_free() (p. 1248)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_close_dup() (p. 1249)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_close_dup_default() (p. 1251)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_close_ocb() (p. 1253)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_close_ocb_default() (p. 1255)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_devctl() (p. 1257)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_devctl_default() (p. 1261)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_devctl_verify() (p. 1263)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_fdinfo() (p. 1266)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_fdinfo_default() (p. 1268)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_func_init() (p. 1271)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_link() (p. 1273)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lock() (p. 1276)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lock_malloc() (p. 1279)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lock_default() (p. 1281)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lock_free() (p. 1283)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lock_ocb_default() (p. 1284)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lseek() (p. 1286)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_lseek_default() (p. 1289)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_mknod() (p. 1291)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_mmap() (p. 1294)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_mmap_default() (p. 1298)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_notify() (p. 1300)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_notify_remove() (p. 1306)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_notify_trigger() (p. 1308)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_ocb_attach() (p. 1311)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_ocb_malloc() (p. 1313)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_ocb_detach() (p. 1315)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
iofunc_ocb_free() (p. 1317)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_open() (p. 1321)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_open_default() (p. 1325)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_openfd() (p. 1327)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_openfd_default() (p. 1330)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_pathconf() (p. 1332)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_pathconf_default() (p. 1335)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_read_default() (p. 1337)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_read_verify() (p. 1339)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_readlink() (p. 1343)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_rename() (p. 1346)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_space_verify() (p. 1350)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_stat() (p. 1353)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_stat_default() (p. 1355)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_sync() (p. 1358)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_sync_default() (p. 1360)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_sync_verify() (p. 1362)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_time_update() (p. 1364)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_unblock() (p. 1366)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_unblock_default() (p. 1368)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_unlink() (p. 1370)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_unlock_ocb_default() (p. 1373)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_utime() (p. 1375)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_utime_default() (p. 1378)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_write_default() (p. 1380)	QNX Neutrino (p. 46)	No	No	Yes	Yes
iofunc_write_verify() (p. 1382)	QNX Neutrino (p. 46)	No	No	No	Yes
ionotify() (p. 1386)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ipsec_dump_policy() (p. 1402)	Unix (p. 46)	No	No	No	Yes
ipsec_get_policylen() (p. 1404)	Unix (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
ipsec_set_policy() (p. 1406)	Unix (p. 46)	No	No	No	Yes
ipsec_strerror() (p. 1410)	Unix (p. 46)	No	Yes	Yes	No
iruserok() (p. 1419)	Unix (p. 46)	Yes	No	No	No
iruserok_sa() (p. 1419)	Unix (p. 46)	Yes	No	No	No
isalnum() (p. 1421)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isalpha() (p. 1423)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iscntrl() (p. 1425)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
isatty() (p. 1427)	POSIX 1003.1 (p. 46)	No	No	No	Yes
isblank() (p. 1429)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iscntrl() (p. 1431)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isdigit() (p. 1433)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isfdtype() (p. 1435)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
isfinite() (p. 1437)	POSIX 1003.1 (p. 46)	No	No	No	Yes
isgraph() (p. 1439)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isinf() (p. 1441)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
islower() (p. 1443)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isnan() (p. 1445)	POSIX 1003.1 (p. 46)	No	No	No	Yes
isprint() (p. 1447)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
ispunct() (p. 1449)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isspace() (p. 1451)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
isupper() (p. 1453)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswalnum() (p. 1455)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswalpha() (p. 1457)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswblank() (p. 1459)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswcntrl() (p. 1461)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswctype() (p. 1463)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
iswdigit() (p. 1465)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswgraph() (p. 1467)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
iswlower() (p. 1469)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
<i>iswprint()</i> (p. 1471)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
<i>iswpunct()</i> (p. 1473)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
<i>iswspace()</i> (p. 1475)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
<i>iswupper()</i> (p. 1477)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
<i>iswxdigit()</i> (p. 1479)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
<i>isxdigit()</i> (p. 1481)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
<i>itoa()</i> (p. 1483)	QNX 4 (p. 46)	No	Yes	Yes	Yes

J

Function	Classification	Cancel	ISR	Signal	Thread
<i>j0()</i> (p. 1486)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	No	Yes
<i>j0f()</i> (p. 1486)	<i>Unix</i> (p. 46)	No	No	No	Yes
<i>j1()</i> (p. 1488)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	No	Yes
<i>j1f()</i> (p. 1488)	<i>Unix</i> (p. 46)	No	No	No	Yes
<i>jn()</i> (p. 1490)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	No	Yes
<i>jnf()</i> (p. 1490)	<i>Unix</i> (p. 46)	No	No	No	Yes
<i>rand48()</i> (p. 1492)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	No	Yes

K

Function	Classification	Cancel	ISR	Signal	Thread
<i>kill()</i> (p. 1494)	<i>POSIX 1003.1</i> (p. 46)	No	No	Yes	Yes
<i>killpg()</i> (p. 1496)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	Yes	Yes

L

Function	Classification	Cancel	ISR	Signal	Thread
labs() (p. 1500)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
lchown() (p. 1502)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
lcong48() (p. 1505)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
ldexp() (p. 1506)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ldexpf() (p. 1506)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ldexpl() (p. 1506)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
ldiv() (p. 1508)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
lldiv() (p. 1508)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
lfind() (p. 1510)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
lgamma() (p. 1513)		No	No	No	Yes
lgamma_r() (p. 1513)		No	No	No	Yes
lgammaf() (p. 1513)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
lgammaf_r() (p. 1513)	QNX Neutrino (p. 46)	No	No	No	Yes
link() (p. 1516)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
lio_listio() (p. 1519)	POSIX 1003.1 AIO (p. 46)	No	No	Yes	Yes
lio_listio64() (p. 1519)	Large-file support (p. 46)	No	No	Yes	Yes
listen() (p. 1523)	POSIX 1003.1 (p. 46)	No	No	No	Yes
llabs() (p. 1525)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
localeconv() (p. 1526)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
localtime() (p. 1531)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	No
localtime_r() (p. 1533)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
lockf() (p. 1535)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
lockf64() (p. 1535)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
log() (p. 1539)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
logf() (p. 1539)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
logl() (p. 1539)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
log10() (p. 1541)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
log10f() (p. 1541)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
log10l() (p. 1541)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
log1p() (p. 1543)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
log1pf() (p. 1543)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
log1pl() (p. 1543)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
logb() (p. 1545)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
logbf() (p. 1545)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
login_tty() (p. 1547)	Unix (p. 46)	Yes	No	Yes	Yes
longjmp() (p. 1549)	ANSI , POSIX 1003.1 (p. 46)	No	No	Yes	Yes
lrand48() (p. 1551)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
lsearch() (p. 1552)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
lseek() (p. 1555)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
lseek64() (p. 1555)	Large-file support (p. 46)	Yes	No	Yes	Yes
lstat() (p. 1558)	POSIX 1003.1 (p. 46)	No	No	Yes	No
lstat64() (p. 1558)	Large-file support (p. 46)	No	No	Yes	No
ltoa() (p. 1560)	QNX 4 (p. 46)	No	Yes	Yes	Yes
lltoa() (p. 1560)	Unix (p. 46)	No	Yes	Yes	Yes
ltrunc() (p. 1562)	QNX 4 (p. 46)	Yes	No	No	Yes

M

Function	Classification	Cancel	ISR	Signal	Thread
main() (p. 1566)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
mallinfo() (p. 1568)	QNX Neutrino (p. 46)	No	No	No	Yes
malloc() (p. 1569)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
mallopt() (p. 1575)	QNX Neutrino (p. 46)	No	No	No	Yes
max() (p. 1582)	QNX 4 (p. 46)	No	Yes	Yes	Yes
mblen() (p. 1583)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mbrlen() (p. 1585)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mbrtowc() (p. 1587)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mbsinit() (p. 1589)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mbsrtowcs() (p. 1591)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mbstowcs() (p. 1593)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mbtowc() (p. 1595)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	No
mcheck() (p. 1598)	QNX Neutrino (p. 46)	No	No	No	Yes
mem_offset() (p. 1600)	QNX Neutrino (p. 46)	No	No	Yes	Yes
mem_offset64() (p. 1600)	QNX Neutrino (p. 46)	No	No	Yes	Yes
memalign() (p. 1603)	QNX Neutrino (p. 46)	No	No	Yes	Yes
memccpy() (p. 1605)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
memchr() (p. 1607)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
memcmp() (p. 1609)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
memcpy() (p. 1611)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
memcpyv() (p. 1613)	QNX 4 (p. 46)	No	Yes	Yes	Yes
memicmp() (p. 1616)	QNX 4 (p. 46)	No	Yes	Yes	Yes
memmove() (p. 1618)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
memset() (p. 1620)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
memset_s() (p. 1622)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
message_attach() (p. 1624)	QNX Neutrino (p. 46)	Yes	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
message_connect() (p. 1630)	QNX Neutrino (p. 46)	Yes	No	No	Yes
message_detach() (p. 1633)	QNX Neutrino (p. 46)	Yes	No	No	Yes
min() (p. 1635)	QNX 4 (p. 46)	No	Yes	Yes	Yes
mkdir() (p. 1636)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mkfifo() (p. 1639)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mknod() (p. 1642)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
mkstemp() (p. 1645)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
mktemp() (p. 1647)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
mktime() (p. 1649)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
mlock() (p. 1651)	POSIX 1003.1 MLR (p. 46)	No	No	Yes	Yes
mlockall() (p. 1653)	POSIX 1003.1 ML (p. 46)	No	No	Yes	Yes
mmap() (p. 1656)	POSIX 1003.1 MFISHMITYM (p. 46)	No	No	Yes	Yes
mmap64() (p. 1656)	Large-file support (p. 46)	No	No	Yes	Yes
mmap_device_io() (p. 1666)	QNX Neutrino (p. 46)	No	No	Yes	Yes
mmap_device_memory() (p. 1668)	QNX Neutrino (p. 46)	No	No	Yes	Yes
modem_open() (p. 1672)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
modem_read() (p. 1676)	QNX Neutrino (p. 46)	Yes	No	Read the Caveats	Read the Caveats
modem_script() (p. 1679)	QNX Neutrino (p. 46)	Yes	No	Read the Caveats	Read the Caveats
modem_write() (p. 1686)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
modf() (p. 1689)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
modff() (p. 1689)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
mount() (p. 1691)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
mount_parse_generic_args() (p. 1694)	QNX Neutrino (p. 46)	No	No	No	Yes
mprobe() (p. 1697)	QNX Neutrino (p. 46)	No	No	No	Yes
mprotect() (p. 1699)	POSIX 1003.1 MPR (p. 46)	No	No	Yes	Yes
mq_close() (p. 1702)	POSIX 1003.1 MSG (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
mq_getattr() (p. 1704)	POSIX 1003.1 MSG (p. 46)	No	No	No	Yes
mq_notify() (p. 1707)	POSIX 1003.1 MSG (p. 46)	No	No	No	Yes
mq_open() (p. 1709)	POSIX 1003.1 MSG (p. 46)	No	No	No	Yes
mq_receive() (p. 1714)	POSIX 1003.1 MSG (p. 46)	Yes	No	No	Yes
mq_send() (p. 1717)	POSIX 1003.1 MSG (p. 46)	Yes	No	No	Yes
mq_setattr() (p. 1720)	POSIX 1003.1 MSG (p. 46)	No	No	No	Yes
mq_timedreceive() (p. 1722)	POSIX 1003.1 MSG (p. 46)	Yes	No	Yes	Yes
mq_timedreceive_monotonic() (p. 1722)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
mq_timedsend() (p. 1725)	POSIX 1003.1 MSG (p. 46)	Yes	No	Yes	Yes
mq_timedsend_monotonic() (p. 1725)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
mq_unlink() (p. 1728)	POSIX 1003.1 MSG (p. 46)	No	No	No	Yes
mrnd48() (p. 1730)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
MsgCurrent() (p. 1734)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgCurrent_r() (p. 1734)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgDeliverEvent() (p. 1736)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgDeliverEvent_r() (p. 1736)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgError() (p. 1743)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgError_r() (p. 1743)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgInfo() (p. 1745)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgInfo_r() (p. 1745)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgKeyData() (p. 1747)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgKeyData_r() (p. 1747)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgRead() (p. 1754)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgRead_r() (p. 1754)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReadv() (p. 1757)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReadv_r() (p. 1757)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceive() (p. 1760)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceive_r() (p. 1760)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceivePulse() (p. 1765)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
MsgReceivePulse_r() (p. 1765)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceivePulsev() (p. 1768)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceivePulsev_r() (p. 1768)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceivev() (p. 1771)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReceivev_r() (p. 1771)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReply() (p. 1775)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReply_r() (p. 1775)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReplyv() (p. 1778)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgReplyv_r() (p. 1778)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSend() (p. 1781)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSend_r() (p. 1781)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendnc() (p. 1785)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendnc_r() (p. 1785)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendPulse() (p. 1789)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendPulse_r() (p. 1789)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendsv() (p. 1793)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendsv_r() (p. 1793)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendsvnc() (p. 1797)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendsvnc_r() (p. 1797)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendv() (p. 1801)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendv_r() (p. 1801)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendvnc() (p. 1805)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendvnc_r() (p. 1805)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendvs() (p. 1809)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendvs_r() (p. 1809)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
MsgSendvsnc() (p. 1813)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgSendvsnc_r() (p. 1813)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgVerifyEvent() (p. 1817)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgVerifyEvent_r() (p. 1817)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
MsgWrite() (p. 1819)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgWrite_r() (p. 1819)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgWritev() (p. 1822)	QNX Neutrino (p. 46)	No	No	Yes	Yes
MsgWritev_r() (p. 1822)	QNX Neutrino (p. 46)	No	No	Yes	Yes
msync() (p. 1825)	POSIX 1003.1 MF SIO (p. 46)	Yes	No	Yes	Yes
munlock() (p. 1828)	POSIX 1003.1 MLR (p. 46)	No	No	Yes	Yes
munlockall() (p. 1830)	POSIX 1003.1 ML (p. 46)	No	No	No	Yes
munmap() (p. 1832)	POSIX 1003.1 MFISHMITYM (p. 46)	No	No	Yes	Yes
munmap_device_io() (p. 1834)	QNX Neutrino (p. 46)	No	No	Yes	Yes
munmap_device_memory() (p. 1836)	QNX Neutrino (p. 46)	No	No	Yes	Yes
munmap_flags() (p. 1838)	QNX Neutrino (p. 46)	No	No	Yes	Yes

N

Function	Classification	Cancel	ISR	Signal	Thread
name_attach() (p. 1842)	QNX Neutrino (p. 46)	Yes	No	No	Yes
name_close() (p. 1848)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
name_detach() (p. 1850)	QNX Neutrino (p. 46)	Yes	No	No	Yes
name_open() (p. 1852)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
nanosleep() (p. 1855)	POSIX 1003.1 TMR (p. 46)	Yes	No	No	Yes
nanospin() (p. 1857)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
nanospin_calibrate() (p. 1859)	QNX Neutrino (p. 46)	No	No	Yes	Yes
nanospin_count() (p. 1861)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
nanospin_ns() (p. 1863)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
nanospin_ns_to_count() (p. 1865)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
nap() (p. 1867)	Unix (p. 46)	Yes	No	Yes	Yes
napms() (p. 1868)	Unix (p. 46)	Yes	No	Yes	Yes
nbaconnect() (p. 1869)	QNX Neutrino (p. 46)	Yes	No	No	Yes
nbaconnect_result() (p. 1871)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ND_NODE_CMP() (p. 1874)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
netmgr_ndtostr() (p. 1877)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
netmgr_remote_nd() (p. 1881)	QNX Neutrino (p. 46)	No	No	Yes	Yes
netmgr_strtond() (p. 1883)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
nextafter() (p. 1885)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
nextafterf() (p. 1885)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
nftw() (p. 1888)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
nftw64() (p. 1888)	Large-file support (p. 46)	Yes	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
<i>nice()</i> (p. 1892)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	Yes	Yes
<i>rand48()</i> (p. 1894)	<i>POSIX 1003.1 XSI</i> (p. 46)	No	No	No	Yes
<i>nsec2timespec()</i> (p. 1895)	<i>QNX Neutrino</i> (p. 46)	No	Yes	Yes	Yes
<i>ntohl()</i> (p. 1896)	<i>POSIX 1003.1</i> (p. 46)	No	Yes	Yes	Yes
<i>ntohs()</i> (p. 1897)	<i>POSIX 1003.1</i> (p. 46)	No	Yes	Yes	Yes

O

Function	Classification	Cancel	ISR	Signal	Thread
offsetof() (p. 1900)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
open() (p. 1905)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
open64() (p. 1905)	Large-file support (p. 46)	Yes	No	Yes	Yes
opendir() (p. 1913)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
openfd() (p. 1916)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
openlog() (p. 1918)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	Yes
openpty() (p. 1921)	Unix (p. 46)	Yes	No	Yes	Yes
out8() (p. 1923)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
out8s() (p. 1925)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
out16() (p. 1927)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
outbe16() (p. 1927)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
outle16() (p. 1927)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
out16s() (p. 1929)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
out32() (p. 1931)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
outbe32() (p. 1931)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
outle32() (p. 1931)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
out32s() (p. 1933)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes

P

Function	Classification	Cancel	ISR	Signal	Thread
pathconf() (p. 1936)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
pathfind() (p. 1939)	Unix (p. 46)	Yes	No	No	No
pathfind() (p. 1939)	Unix (p. 46)	Yes	No	Yes	Yes
pathfind_r() (p. 1939)	Unix (p. 46)	Yes	No	No	No
pathfind_r() (p. 1939)	Unix (p. 46)	Yes	No	Yes	Yes
pathmgr_symlink() (p. 1943)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pathmgr_unlink() (p. 1945)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pause() (p. 1947)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
pccard_arm() (p. 1949)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pccard_attach() (p. 1952)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pccard_detach() (p. 1954)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pccard_info() (p. 1956)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pccard_lock() (p. 1958)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pccard_raw_read() (p. 1960)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pccard_unlock() (p. 1962)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_attach() (p. 1964)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_attach_device() (p. 1966)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_detach() (p. 1975)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_detach_device() (p. 1977)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_find_class() (p. 1979)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_find_device() (p. 1981)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_irq_routing_options() (p. 1983)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_map_irq() (p. 1985)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_present() (p. 1987)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_read_config() (p. 1989)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_read_config8() (p. 1991)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
pci_read_config16() (p. 1993)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_read_config32() (p. 1995)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_rescan_bus() (p. 1997)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_write_config() (p. 1999)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_write_config8() (p. 2001)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_write_config16() (p. 2003)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pci_write_config32() (p. 2005)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pclose() (p. 2007)	POSIX 1003.1 (p. 46)	Yes	No	No	No
perror() (p. 2009)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
pipe() (p. 2011)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
poll() (p. 2013)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	Yes
popen() (p. 2019)	POSIX 1003.1 (p. 46)	Yes	No	No	No
posix_fadvise() (p. 2022)	POSIX 1003.1 ADV (p. 46)	Yes	No	Yes	Yes
posix_fadvise64() (p. 2022)	Large-file support (p. 46)	Yes	No	Yes	Yes
posix_fallocate() (p. 2025)	POSIX 1003.1 ADV (p. 46)	Yes	No	Yes	Yes
posix_fallocate64() (p. 2025)	Large-file support (p. 46)	Yes	No	Yes	Yes
posix_madvise() (p. 2028)	POSIX 1003.1 ADV (p. 46)	No	No	Yes	Yes
posix_mem_offset() (p. 2030)	POSIX 1003.1 TYM; (p. 46)	No	No	Yes	Yes
posix_mem_offset64() (p. 2030)	large-file support (p. 46)	No	No	Yes	Yes
posix_memalign() (p. 2033)	POSIX 1003.1 ADV (p. 46)	No	No	Yes	Yes
posix_openpt() (p. 2035)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
posix_spawn() (p. 2037)	POSIX 1003.1 SPN (p. 46)	Yes	No	Yes	Yes
posix_spawnnp() (p. 2037)	POSIX 1003.1 SPN (p. 46)	Yes	No	Yes	Yes
posix_spawn_file_actions_addclose() (p. 2048)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
posix_spawn_file_actions_adddup2() (p. 2050)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
posix_spawn_file_actions_addopen() (p. 2052)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
<i>posix_spawn_file_actions_destroy()</i> (p. 2055)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawn_file_actions_init()</i> (p. 2057)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_addpartid()</i> (p. 2059)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_addpartition()</i> (p. 2061)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_destroy()</i> (p. 2064)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getcred()</i> (p. 2066)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getflags()</i> (p. 2068)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getnode()</i> (p. 2070)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getpartid()</i> (p. 2072)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getpgroup()</i> (p. 2074)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getrunmask()</i> (p. 2076)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getschedparam()</i> (p. 2078)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getschedpolicy()</i> (p. 2080)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getsigdefault()</i> (p. 2082)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getsigignore()</i> (p. 2084)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getsigmask()</i> (p. 2086)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getstackmax()</i> (p. 2088)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_getxflags()</i> (p. 2090)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_init()</i> (p. 2093)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_setcred()</i> (p. 2095)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_setflags()</i> (p. 2098)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_setnode()</i> (p. 2101)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_setpgroup()</i> (p. 2103)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_setrunmask()</i> (p. 2105)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
<i>posix_spawnattr_setschedparam()</i> (p. 2107)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
posix_spawnattr_setschedpolicy() (p. 2109)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
posix_spawnattr_setsigdefault() (p. 2111)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
posix_spawnattr_setsigignore() (p. 2113)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
posix_spawnattr_setsigmask() (p. 2115)	POSIX 1003.1 SPN (p. 46)	No	Yes	Yes	Yes
posix_spawnattr_setstackmax() (p. 2117)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
posix_spawnattr_setxflags() (p. 2119)	QNX Neutrino (p. 46)	No	No	No	Yes
posix_typed_mem_get_info() (p. 2122)	POSIX 1003.1 TYM (p. 46)	No	No	Yes	Yes
posix_typed_mem_open() (p. 2124)	POSIX 1003.1 TYM (p. 46)	No	No	Yes	Yes
pow() (p. 2129)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
powf() (p. 2129)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
powl() (p. 2129)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
pread() (p. 2131)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
pread64() (p. 2131)	Large-file support (p. 46)	Yes	No	Yes	Yes
printf() (p. 2133)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
procmgr_ability() (p. 2141)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_ability_create() (p. 2152)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_ability_lookup() (p. 2154)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_daemon() (p. 2156)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_event_notify() (p. 2158)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_event_notify_add() (p. 2163)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_event_notify_delete() (p. 2168)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_event_trigger() (p. 2170)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_guardian() (p. 2172)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_session() (p. 2175)	QNX Neutrino (p. 46)	No	No	No	Yes
procmgr_timer_tolerance() (p. 2178)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_value_current() (p. 2180)	QNX Neutrino (p. 46)	No	No	Yes	Yes
procmgr_value_notify_add() (p. 2181)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_abort() (p. 2185)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
pthread_atfork() (p. 2187)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_destroy() (p. 2189)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_getdetachstate() (p. 2191)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_getguardsize() (p. 2193)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
pthread_attr_getinheritsched() (p. 2195)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_attr_getschedparam() (p. 2197)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_getschedpolicy() (p. 2199)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_attr_getscope() (p. 2201)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_attr_getstack() (p. 2203)	POSIX 1003.1 TSA TSS (p. 46)	No	No	Yes	Yes
pthread_attr_getstackaddr() (p. 2205)	POSIX 1003.1 THR TSA (p. 46)	No	No	Yes	Yes
pthread_attr_getstacklazy() (p. 2207)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_attr_getstackprealloc() (p. 2209)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_attr_getstacksize() (p. 2211)	POSIX 1003.1 THR TSS (p. 46)	No	No	Yes	Yes
pthread_attr_init() (p. 2213)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_setdetachstate() (p. 2215)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_setguardsize() (p. 2217)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
pthread_attr_setinheritsched() (p. 2219)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_attr_setschedparam() (p. 2221)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_attr_setschedpolicy() (p. 2223)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_attr_setscope() (p. 2225)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_attr_setstack() (p. 2227)	POSIX 1003.1 TSA TSS (p. 46)	No	No	Yes	Yes
pthread_attr_setstackaddr() (p. 2229)	POSIX 1003.1 THR TSA (p. 46)	No	No	Yes	Yes
pthread_attr_setstacklazy() (p. 2231)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_attr_setstackprealloc() (p. 2233)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_attr_setstacksize() (p. 2235)	POSIX 1003.1 THR TSS (p. 46)	No	No	Yes	Yes
pthread_barrier_destroy() (p. 2237)	POSIX 1003.1 THR BAR (p. 46)	No	No	Yes	Yes
pthread_barrier_init() (p. 2239)	POSIX 1003.1 THR BAR (p. 46)	No	No	Yes	Yes
pthread_barrier_wait() (p. 2241)	POSIX 1003.1 THR BAR (p. 46)	No	No	Yes	Yes
pthread_barrierattr_destroy() (p. 2243)	POSIX 1003.1 THR BAR (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
<i>pthread_barrierattr_getpshared()</i> (p. 2245)	POSIX 1003.1 THR BAR TSH (p. 46)	No	No	Yes	Yes
<i>pthread_barrierattr_init()</i> (p. 2247)	POSIX 1003.1 THR BAR (p. 46)	No	No	Yes	Yes
<i>pthread_barrierattr_setpshared()</i> (p. 2249)	POSIX 1003.1 THR BAR TSH (p. 46)	No	No	Yes	Yes
<i>pthread_cancel()</i> (p. 2251)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cleanup_pop()</i> (p. 2253)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cleanup_push()</i> (p. 2255)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cond_broadcast()</i> (p. 2257)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cond_destroy()</i> (p. 2259)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cond_init()</i> (p. 2261)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cond_signal()</i> (p. 2263)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_cond_timedwait()</i> (p. 2265)	POSIX 1003.1 THR (p. 46)	Yes	No	Yes	Yes
<i>pthread_cond_wait()</i> (p. 2269)	POSIX 1003.1 THR (p. 46)	Yes	No	Yes	Yes
<i>pthread_condattr_destroy()</i> (p. 2272)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_condattr_getclock()</i> (p. 2274)	POSIX 1003.1 THR CS (p. 46)	No	No	Yes	Yes
<i>pthread_condattr_getpshared()</i> (p. 2276)	POSIX 1003.1 THR TSH (p. 46)	No	No	Yes	Yes
<i>pthread_condattr_init()</i> (p. 2278)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_condattr_setclock()</i> (p. 2280)	POSIX 1003.1 THR CS (p. 46)	No	No	Yes	Yes
<i>pthread_condattr_setpshared()</i> (p. 2282)	POSIX 1003.1 THR TSH (p. 46)	No	No	Yes	Yes
<i>pthread_create()</i> (p. 2284)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_detach()</i> (p. 2288)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_equal()</i> (p. 2290)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_exit()</i> (p. 2292)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_getconcurrency()</i> (p. 2294)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
<i>pthread_getcpuclockid()</i> (p. 2295)	POSIX 1003.1 THR TCT (p. 46)	No	No	Yes	Yes
<i>pthread_getname_np()</i> (p. 2297)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
<i>pthread_getschedparam()</i> (p. 2299)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
<i>pthread_getspecific()</i> (p. 2301)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
<i>pthread_join()</i> (p. 2303)	POSIX 1003.1 THR (p. 46)	Yes	No	Yes	Yes
<i>pthread_key_create()</i> (p. 2305)	POSIX 1003.1 THR (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
pthread_key_delete() (p. 2308)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_kill() (p. 2310)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutex_destroy() (p. 2312)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutex_getprioceiling() (p. 2314)	POSIX 1003.1 THR TPP (p. 46)	No	No	Yes	Yes
pthread_mutex_init() (p. 2316)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutex_lock() (p. 2318)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutex_setprioceiling() (p. 2322)	POSIX 1003.1 THR TPP (p. 46)	No	No	Yes	Yes
pthread_mutex_timedlock() (p. 2324)	POSIX 1003.1 THR TMO (p. 46)	No	No	Yes	Yes
pthread_mutex_timedlock_monotonic() (p. 2324)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_mutex_trylock() (p. 2327)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutex_unlock() (p. 2329)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutex_wakeup_np() (p. 2331)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_mutexattr_destroy() (p. 2333)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutexattr_getprioceiling() (p. 2335)	POSIX 1003.1 THR TPP (p. 46)	No	No	Yes	Yes
pthread_mutexattr_getprotocol() (p. 2337)	POSIX 1003.1 THR TPPITPI (p. 46)	No	No	Yes	Yes
pthread_mutexattr_getpshared() (p. 2339)	POSIX 1003.1 THR TSH (p. 46)	No	No	Yes	Yes
pthread_mutexattr_getrecursive() (p. 2341)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_mutexattr_gettype() (p. 2343)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
pthread_mutexattr_getwakeup_np() (p. 2345)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
pthread_mutexattr_init() (p. 2347)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_mutexattr_setprioceiling() (p. 2349)	POSIX 1003.1 THR TPP (p. 46)	No	No	Yes	Yes
pthread_mutexattr_setprotocol() (p. 2351)	POSIX 1003.1 THR TPPITPI (p. 46)	No	No	Yes	Yes
pthread_mutexattr_setpshared() (p. 2353)	POSIX 1003.1 THR TSH (p. 46)	No	No	Yes	Yes
pthread_mutexattr_setrecursive() (p. 2355)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
pthread_mutexattr_settype() (p. 2357)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
pthread_mutexattr_setwakeup_np() (p. 2359)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
pthread_once() (p. 2361)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlock_destroy() (p. 2364)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlock_init() (p. 2366)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlock_rdlock() (p. 2369)	POSIX 1003.1 THR (p. 46)	Yes	No	Yes	Yes
pthread_rwlock_timedrdlock() (p. 2371)	POSIX 1003.1 THR TMO (p. 46)	Yes	No	Yes	Yes
pthread_rwlock_timedwrlock() (p. 2373)	POSIX 1003.1 THR TMO (p. 46)	Yes	No	Yes	Yes
pthread_rwlock_tryrdlock() (p. 2375)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlock_trywrlock() (p. 2377)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlock_unlock() (p. 2379)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlock_wrlock() (p. 2381)	POSIX 1003.1 THR (p. 46)	Yes	No	Yes	Yes
pthread_rwlockattr_destroy() (p. 2383)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlockattr_getpshared() (p. 2385)	POSIX 1003.1 THR TSH (p. 46)	No	No	Yes	Yes
pthread_rwlockattr_init() (p. 2387)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_rwlockattr_setpshared() (p. 2389)	POSIX 1003.1 THR TSH (p. 46)	No	No	Yes	Yes
pthread_self() (p. 2391)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_setcancelstate() (p. 2392)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_setcanceltype() (p. 2394)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_setconcurrency() (p. 2396)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
pthread_setname_np() (p. 2398)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pthread_setschedparam() (p. 2400)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_setschedprio() (p. 2402)	POSIX 1003.1 THR TPS (p. 46)	No	No	Yes	Yes
pthread_setspecific() (p. 2405)	POSIX 1003.1 THR (p. 46)	No	No	No	Yes
pthread_sigmask() (p. 2407)	POSIX 1003.1 THR (p. 46)	No	No	Yes	Yes
pthread_sleepon_broadcast() (p. 2409)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_sleepon_lock() (p. 2411)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pthread_sleepon_signal() (p. 2413)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
pthread_sleepon_timedwait() (p. 2415)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pthread_sleepon_unlock() (p. 2418)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_sleepon_wait() (p. 2419)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
pthread_spin_destroy() (p. 2422)	POSIX 1003.1 THR SPI (p. 46)	No	No	Yes	Yes
pthread_spin_init() (p. 2424)	POSIX 1003.1 THR SPI (p. 46)	No	No	Yes	Yes
pthread_spin_lock() (p. 2426)	POSIX 1003.1 THR SPI (p. 46)	No	No	Yes	Yes
pthread_spin_trylock() (p. 2428)	POSIX 1003.1 THR SPI (p. 46)	No	No	Yes	Yes
pthread_spin_unlock() (p. 2430)	POSIX 1003.1 THR SPI (p. 46)	No	No	Yes	Yes
pthread_testcancel() (p. 2432)	POSIX 1003.1 THR (p. 46)	Yes	No	Yes	Yes
pthread_timedjoin() (p. 2433)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pthread_timedjoin_monotonic() (p. 2433)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ptsname() (p. 2436)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	No
ptsname_r() (p. 2438)	QNX Neutrino (p. 46)	No	No	Yes	Yes
pulse_attach() (p. 2442)	QNX Neutrino (p. 46)	Yes	No	No	Yes
pulse_detach() (p. 2445)	QNX Neutrino (p. 46)	No	No	No	Yes
putc() (p. 2447)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
putc_unlocked() (p. 2449)	POSIX 1003.1 TSF (p. 46)	Yes	No	No	No
putchar() (p. 2451)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
putchar_unlocked() (p. 2453)	POSIX 1003.1 TSF (p. 46)	Yes	No	No	No
putenv() (p. 2454)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
puts() (p. 2457)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
putspent() (p. 2459)	Unix (p. 46)	Yes	No	No	No
pututline() (p. 2462)	Unix (p. 46)	Yes	No	No	Yes
putw() (p. 2464)	Legacy Unix (p. 46)	Yes	No	Yes	Yes
putwc() (p. 2466)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
putwchar() (p. 2468)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
pwrite() (p. 2470)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
pwrite64() (p. 2470)	Large-file support (p. 46)	Yes	No	Yes	Yes

Q

Function	Classification	Cancel	ISR	Signal	Thread
<i>qnx_crypt()</i> (p. 2474)	<i>QNX Neutrino</i> (p. 46)	No	No	Yes	Yes
<i>qsort()</i> (p. 2476)	<i>ANSI</i> (p. 46) <i>POSIX 1003.1</i> (p. 46)	No	No	No	Yes

R

Function	Classification	Cancel	ISR	Signal	Thread
Raccept() (p. 2480)	SOCKS (p. 46)	No	No	No	No
raise() (p. 2482)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
rand() (p. 2484)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	No
rand_r() (p. 2486)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
random() (p. 2487)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	No
Rbind() (p. 2489)	SOCKS (p. 46)	No	No	No	No
rcmd() (p. 2491)	Unix (p. 46)	Yes	No	No	No
rcmd_af() (p. 2491)	Unix (p. 46)	Yes	No	No	No
Rconnect() (p. 2494)	SOCKS (p. 46)	No	No	No	No
rdchk() (p. 2496)	Unix (p. 46)	No	No	Yes	Yes
re_comp() (p. 2498)	Legacy Unix (p. 46)	No	No	Yes	Yes
re_exec() (p. 2500)	Legacy Unix (p. 46)	No	No	Yes	Yes
read() (p. 2502)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
readblock() (p. 2506)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
readcond() (p. 2509)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
readdir() (p. 2514)	POSIX 1003.1 (p. 46)	Yes	No	No	No
readdir64() (p. 2514)	Large-file support (p. 46)	Yes	No	No	No
readdir_r() (p. 2518)	POSIX 1003.1 TSF (p. 46)	Yes	No	Yes	Yes
readlink() (p. 2520)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
readv() (p. 2523)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
realloc() (p. 2527)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
realpath() (p. 2531)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
recv() (p. 2533)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
recvfrom() (p. 2536)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
recvmsg() (p. 2539)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
regcomp() (p. 2542)	POSIX 1003.1 (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
regerror() (p. 2547)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
regexec() (p. 2549)	POSIX 1003.1 (p. 46)	No	No	No	Yes
regfree() (p. 2552)	POSIX 1003.1 (p. 46)	No	No	No	Yes
remainder() (p. 2554)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
remainderf() (p. 2554)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
remainderl() (p. 2554)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
remove() (p. 2556)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
remque() (p. 2559)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
rename() (p. 2560)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
res_init() (p. 2563)	Unix (p. 46)	Yes	No	No	No
res_mkquery() (p. 2566)	Unix (p. 46)	Yes	No	No	No
res_query() (p. 2569)	Unix (p. 46)	Yes	No	No	No
res_querydomain() (p. 2572)	Unix (p. 46)	Yes	No	No	No
res_search() (p. 2575)	Unix (p. 46)	Yes	No	No	No
res_send() (p. 2578)	Unix (p. 46)	Yes	No	No	No
resmgr_attach() (p. 2580)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_block() (p. 2589)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_context_alloc() (p. 2594)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_context_free() (p. 2596)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_detach() (p. 2600)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_devino() (p. 2603)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_handle_grow() (p. 2605)	QNX Neutrino (p. 46)	No	No	No	Yes
resmgr_handle_tune() (p. 2606)	QNX Neutrino (p. 46)	No	No	No	Yes
resmgr_handler() (p. 2608)	QNX Neutrino (p. 46)	Yes	No	No	Yes
resmgr_iofuncs() (p. 2614)	QNX Neutrino (p. 46)	No	No	No	Yes
resmgr_msg_again() (p. 2616)	QNX Neutrino (p. 46)	No	No	No	Yes
resmgr_msgread() (p. 2618)	QNX Neutrino (p. 46)	No	No	Yes	Yes
resmgr_msgreadv() (p. 2620)	QNX Neutrino (p. 46)	No	No	Yes	Yes
resmgr_msgreply() (p. 2622)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
resmgr_msgreplyv() (p. 2625)	QNX Neutrino (p. 46)	No	No	Yes	Yes
resmgr_msgwrite() (p. 2628)	QNX Neutrino (p. 46)	No	No	Yes	Yes
resmgr_msgwritev() (p. 2630)	QNX Neutrino (p. 46)	No	No	Yes	Yes
resmgr_ocb() (p. 2634)	QNX Neutrino (p. 46)	No	No	No	Yes
resmgr_open_bind() (p. 2636)	QNX Neutrino (p. 46)	No	No	No	Yes
resmgr_pathname() (p. 2639)	QNX Neutrino (p. 46)	No	No	Yes	Yes
resmgr_unbind() (p. 2644)	QNX Neutrino (p. 46)	No	No	No	Yes
rewind() (p. 2646)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
rewinddir() (p. 2648)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
Rgetsockname() (p. 2650)	SOCKS (p. 46)	No	No	No	No
rindex() (p. 2652)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
rint() (p. 2654)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
rintf() (p. 2654)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
rintl() (p. 2654)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
Rlisten() (p. 2656)	SOCKS (p. 46)	No	No	No	No
rmdir() (p. 2658)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
round() (p. 2661)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
roundf() (p. 2661)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
roundl() (p. 2661)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
Rrcmd() (p. 2668)	SOCKS (p. 46)	No	No	No	No
rresvport() (p. 2670)	Unix (p. 46)	Yes	No	No	Yes
rresvport_af() (p. 2670)	Unix (p. 46)	Yes	No	No	Yes
Rselect() (p. 2672)	SOCKS (p. 46)	No	No	No	No
rsrdbmgr_attach() (p. 2674)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
rsrdbmgr_create() (p. 2681)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
rsrdbmgr_destroy() (p. 2684)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
rsrdbmgr_detach() (p. 2686)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
rsrdbmgr_devno_attach() (p. 2688)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
rsrdbmgr_devno_detach() (p. 2691)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
rsrdbmgr_query_name() (p. 2693)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ruserok() (p. 2696)	Unix (p. 46)	Yes	No	No	No

S

Function	Classification	Cancel	ISR	Signal	Thread
sbrk() (p. 2702)	Legacy Unix (p. 46)	No	No	No	Yes
scalb() (p. 2705)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
scalbn() (p. 2707)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
scalbnf() (p. 2707)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
scandir() (p. 2711)	Legacy Unix (p. 46)	Yes	No	Yes	Yes
scanf() (p. 2713)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
sched_getparam() (p. 2720)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_get_priority_adjust() (p. 2723)	QNX Neutrino (p. 46)	No	No	Yes	Yes
sched_get_priority_max() (p. 2725)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_get_priority_min() (p. 2727)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_getscheduler() (p. 2729)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_rr_get_interval() (p. 2736)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_setparam() (p. 2738)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_setscheduler() (p. 2740)	POSIX 1003.1 PS (p. 46)	No	No	Yes	Yes
sched_yield() (p. 2743)	POSIX 1003.1 PS/THR (p. 46)	No	No	Yes	Yes
SchedCtl() (p. 2745)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedCtl_r() (p. 2745)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedGet() (p. 2774)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedGet_r() (p. 2774)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedInfo() (p. 2777)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedInfo_r() (p. 2777)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedSet() (p. 2780)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedSet_r() (p. 2780)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedYield() (p. 2783)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SchedYield_r() (p. 2783)	QNX Neutrino (p. 46)	No	No	Yes	Yes
searchenv() (p. 2785)	QNX 4 (p. 46)	No	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
seed48() (p. 2787)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
seekdir() (p. 2789)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
select() (p. 2791)	POSIX 1003.1 (p. 46)	Yes	No	No	Read the Caveats
select_attach() (p. 2796)	QNX Neutrino (p. 46)	Yes	No	No	Yes
select_detach() (p. 2800)	QNX Neutrino (p. 46)	Yes	No	No	Yes
select_query() (p. 2802)	QNX Neutrino (p. 46)	No	No	No	Yes
sem_close() (p. 2805)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_destroy() (p. 2807)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_getvalue() (p. 2809)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_init() (p. 2811)	POSIX 1003.1 SEM (p. 46)	No	No	No	Yes
sem_open() (p. 2814)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_post() (p. 2818)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_timedwait() (p. 2820)	POSIX 1003.1 SEM TMO (p. 46)	Yes	No	Yes	Yes
sem_timedwait_monotonic() (p. 2820)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
sem_trywait() (p. 2823)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_unlink() (p. 2825)	POSIX 1003.1 SEM (p. 46)	No	No	Yes	Yes
sem_wait() (p. 2827)	POSIX 1003.1 SEM (p. 46)	Yes	No	Yes	Yes
send() (p. 2829)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
sendmsg() (p. 2833)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
sendto() (p. 2837)	POSIX 1003.1 (p. 46)	Yes	No	No	Yes
set_lowest_fd() (p. 2843)	QNX Neutrino (p. 46)	No	No	No	No
setbuf() (p. 2845)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
setbuffer() (p. 2847)	Unix (p. 46)	No	No	No	Yes
setdomainname() (p. 2849)	Unix (p. 46)	No	No	Yes	Yes
setegid() (p. 2851)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
setenv() (p. 2853)	POSIX 1003.1 (p. 46)	No	No	No	No
seteuid() (p. 2856)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
setgid() (p. 2858)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
setfsent() (p. 2860)	NetBSD (p. 46)	Yes	No	No	No
setgrent() (p. 2862)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
setgroups() (p. 2864)	Unix (p. 46)	Yes	No	No	No
sethostent() (p. 2866)	POSIX 1003.1 (p. 46)	Yes	No	No	No
sethostname() (p. 2868)	Unix (p. 46)	Yes	No	Yes	Yes
SETIOV() (p. 2870)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
setitimer() (p. 2872)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
setjmp() (p. 2875)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
setkey() (p. 2877)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
setlinebuf() (p. 2878)	Unix (p. 46)	No	No	No	Yes
setlocale() (p. 2880)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
setlogmask() (p. 2882)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
setnetent() (p. 2884)	POSIX 1003.1 (p. 46)	Yes	No	No	No
setpgid() (p. 2886)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
setpgrp() (p. 2889)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
setprio() (p. 2890)	QNX 4 (p. 46)	No	No	Yes	Yes
setprotoent() (p. 2892)	POSIX 1003.1 (p. 46)	Yes	No	No	No
setpwent() (p. 2894)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
setregid() (p. 2895)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
setreuid() (p. 2897)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
setrlimit() (p. 2899)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
setrlimit64() (p. 2899)	Large-file support (p. 46)	No	No	Yes	Yes
setservent() (p. 2905)	POSIX 1003.1 (p. 46)	Yes	No	No	No
setsid() (p. 2907)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
setsockopt() (p. 2909)	POSIX 1003.1 (p. 46)	No	No	No	Yes
setspent() (p. 2912)	Unix (p. 46)	No	No	No	No
setstate() (p. 2913)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
settimeofday() (p. 2915)	Legacy Unix (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
setuid() (p. 2917)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
setutent() (p. 2919)	Unix (p. 46)	Yes	No	Yes	Yes
setvbuf() (p. 2920)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
shm_ctl() (p. 2925)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
shm_ctl_special() (p. 2931)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
shm_open() (p. 2935)	POSIX 1003.1 SHM (p. 46)	No	No	Yes	Yes
shm_unlink() (p. 2940)	POSIX 1003.1 SHM (p. 46)	No	No	Yes	Yes
shutdown() (p. 2942)	POSIX 1003.1 (p. 46)	No	No	No	Yes
shutdown_system() (p. 2944)	QNX Neutrino (p. 46)	No	No	No	No
sigaction() (p. 2948)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
sigaddset() (p. 2953)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
sigblock() (p. 2955)	Unix (p. 46)	No	No	Yes	No
sigdelset() (p. 2957)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
sigemptyset() (p. 2959)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
sigfillset() (p. 2966)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
sigismember() (p. 2968)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
siglongjmp() (p. 2970)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
sigmask() (p. 2972)	Unix (p. 46)	No	Yes	Yes	Yes
signal() (p. 2974)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
SignalAction() (p. 2977)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SignalAction_r() (p. 2977)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SignalKill() (p. 2986)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SignalKill_r() (p. 2986)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SignalProcmask() (p. 2991)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SignalProcmask_r() (p. 2991)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SignalSuspend() (p. 2995)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SignalSuspend_r() (p. 2995)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SignalWaitinfo() (p. 2997)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SignalWaitinfo_r() (p. 2997)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
sigpause() (p. 3000)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
sigpending() (p. 3002)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
sigprocmask() (p. 3004)	POSIX 1003.1 CX (p. 46)	No	No	Yes	Yes
sigqueue() (p. 3007)	POSIX 1003.1 RTS (p. 46)	No	No	Yes	Yes
sigsetjmp() (p. 3010)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
sigsetmask() (p. 3012)	Unix (p. 46)	No	No	Yes	Yes
sigsuspend() (p. 3014)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
sigtimedwait() (p. 3016)	POSIX 1003.1 RTS (p. 46)	Yes	No	Yes	Yes
sigunblock() (p. 3018)	Unix (p. 46)	No	No	Yes	Yes
sigwait() (p. 3020)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
sigwaitinfo() (p. 3022)	POSIX 1003.1 RTS (p. 46)	Yes	No	Yes	Yes
sin() (p. 3024)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sinf() (p. 3024)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sinl() (p. 3024)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sinh() (p. 3026)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sinhf() (p. 3026)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sinhl() (p. 3026)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sleep() (p. 3028)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
slog2c() (p. 3044)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
slog2f() (p. 3046)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
slog2fa() (p. 3048)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
slog2_get_verbosity() (p. 3051)	QNX Neutrino (p. 46)	No	No	Yes	Yes
slog2_register() (p. 3052)	QNX Neutrino (p. 46)	No	No	Yes	Yes
slog2_reset() (p. 3056)	QNX Neutrino (p. 46)	No	No	Yes	Yes
slog2_set_default_buffer() (p. 3057)	QNX Neutrino (p. 46)	No	No	Yes	Yes
slog2_set_verbosity() (p. 3059)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
slogb() (p. 3061)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
slogf() (p. 3063)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
slogi() (p. 3066)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
snprintf() (p. 3070)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
socketmark() (p. 3073)	POSIX 1003.1 (p. 46)	No	No	No	No
socket() (p. 3075)	POSIX 1003.1 (p. 46)	No	No	No	Yes
socketpair() (p. 3078)	POSIX 1003.1 (p. 46)	No	No	No	Yes
SOCKSinit() (p. 3081)	SOCKS (p. 46)	No	No	No	No
sopen() (p. 3083)	Unix (p. 46)	Yes	No	No	Yes
sopenfd() (p. 3087)	Unix (p. 46)	Yes	No	Yes	Yes
spawn() (p. 3089)	QNX Neutrino (p. 46)	No	No	Yes	Yes
spawnl() (p. 3099)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
spawnle() (p. 3105)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
spawnlp() (p. 3111)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
spawnlpe() (p. 3116)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
spawnp() (p. 3122)	QNX Neutrino (p. 46)	No	No	No	Yes
spawnv() (p. 3129)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
spawnve() (p. 3134)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
spawnvp() (p. 3140)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
spawnvpe() (p. 3145)	QNX 4 (p. 46)	Read the Caveats	No	No	Yes
sprintf() (p. 3151)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
sqrt() (p. 3153)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sqrtf() (p. 3153)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
sqrtl() (p. 3153)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
srand() (p. 3155)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
srand48() (p. 3157)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
srandom() (p. 3158)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
sscanf() (p. 3162)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
stat() (p. 3164)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
stat64() (p. 3164)	Large-file support (p. 46)	No	No	Yes	Yes
statvfs() (p. 3170)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
statvfs64() (p. 3170)	Large-file support (p. 46)	No	No	Yes	Yes
straddstr() (p. 3178)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
strcasecmp() (p. 3180)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
strcat() (p. 3182)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strchr() (p. 3184)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strcmp() (p. 3186)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strcmpi() (p. 3188)	QNX 4 (p. 46)	No	Yes	Yes	Yes
strcoll() (p. 3190)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strcpy() (p. 3192)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strcspn() (p. 3194)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
strdup() (p. 3196)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
strerror() (p. 3198)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
strerror_r() (p. 3200)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
strftime() (p. 3202)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
stricmp() (p. 3208)	QNX 4 (p. 46)	No	Yes	Yes	Yes
strlcat() (p. 3210)	Unix (p. 46)	No	Yes	Yes	Yes
strlcpy() (p. 3210)	Unix (p. 46)	No	Yes	Yes	Yes
strlen() (p. 3213)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strlwr() (p. 3215)	Unix (p. 46)	No	Yes	Yes	Yes
strncasecmp() (p. 3217)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
strncat() (p. 3219)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strncmp() (p. 3221)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strncpy() (p. 3223)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strnicmp() (p. 3225)	QNX 4 (p. 46)	No	Yes	Yes	Yes
strnlen() (p. 3227)	POSIX 1003.1-2008 CX (p. 46)	No	Yes	Yes	Yes
strnset() (p. 3229)	QNX 4 (p. 46)	No	Yes	Yes	Yes
strpbrk() (p. 3231)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strptime() (p. 3233)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
strrchr() (p. 3239)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strrev() (p. 3241)	QNX 4 (p. 46)	No	Yes	Yes	Yes
strsep() (p. 3243)	Unix (p. 46)	No	Yes	Yes	Yes
strset() (p. 3245)	QNX 4 (p. 46)	No	Yes	Yes	Yes
strsignal() (p. 3247)	Unix (p. 46)	No	No	Yes	Yes
strspn() (p. 3249)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strstr() (p. 3251)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strtod() (p. 3253)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
strtof() (p. 3253)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
strtold() (p. 3253)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
strtoimax() (p. 3256)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
strtoumax() (p. 3256)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strtok() (p. 3258)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	No
strtok_r() (p. 3260)	POSIX 1003.1 TSF (p. 46)	No	Yes	Yes	Yes
strtol() (p. 3262)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strtoll() (p. 3262)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strtoul() (p. 3264)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strtoull() (p. 3264)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
strupr() (p. 3266)	Unix (p. 46)	No	Yes	Yes	Yes
strxfrm() (p. 3268)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
swab() (p. 3270)	POSIX 1003.1 XSI (p. 46)	No	Yes	Yes	Yes
swprintf() (p. 3272)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
swscanf() (p. 3274)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
symlink() (p. 3276)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
sync() (p. 3279)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
SyncCondvarSignal() (p. 3280)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SyncCondvarSignal_r() (p. 3280)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SyncCondvarWait() (p. 3282)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SyncCondvarWait_r() (p. 3282)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
SyncCtl() (p. 3286)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncCtl_r() (p. 3286)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncDestroy() (p. 3289)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncDestroy_r() (p. 3289)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexEvent() (p. 3291)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexEvent_r() (p. 3291)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexLock() (p. 3294)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexLock_r() (p. 3294)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
SyncMutexRevive() (p. 3297)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexRevive_r() (p. 3297)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexUnlock() (p. 3299)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncMutexUnlock_r() (p. 3299)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncSemPost() (p. 3301)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncSemPost_r() (p. 3301)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncSemWait() (p. 3303)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncSemWait_r() (p. 3303)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncTypeCreate() (p. 3305)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SyncTypeCreate_r() (p. 3305)	QNX Neutrino (p. 46)	No	No	Yes	Yes
sysconf() (p. 3308)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
sysctl() (p. 3311)	Unix (p. 46)	Yes	No	No	Yes
syslog() (p. 3316)	POSIX 1003.1 XSI (p. 46)	Yes	No	No	No
sysmgr_reboot() (p. 3319)	QNX Neutrino (p. 46)	No	No	Yes	Yes
sysmgr_runstate() (p. 3320)	QNX Neutrino (p. 46)	No	No	Yes	Yes
sysmgr_runstate_dynamic() (p. 3320)	QNX Neutrino (p. 46)	No	No	Yes	Yes
sysmgr_runstate_burst() (p. 3322)	QNX Neutrino (p. 46)	No	No	Yes	Yes
SYSPAGE_CPU_ENTRY() (p. 3324)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
SYSPAGE_ENTRY() (p. 3326)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
system() (p. 3330)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes

T

Function	Classification	Cancel	ISR	Signal	Thread
tan() (p. 3334)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
tanf() (p. 3334)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
tanl() (p. 3334)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
tanh() (p. 3336)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
tanhf() (p. 3336)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
tanhl() (p. 3336)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
tcdrain() (p. 3338)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
tcdropline() (p. 3340)	QNX 4 (p. 46)	No	No	Yes	Yes
tcflow() (p. 3342)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcflush() (p. 3345)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcgetattr() (p. 3347)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcgetpgrp() (p. 3349)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcgetsid() (p. 3351)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
tcgetsize() (p. 3353)	QNX Neutrino (p. 46)	No	No	Yes	Yes
tcinject() (p. 3355)	QNX Neutrino (p. 46)	No	No	Yes	Yes
tcischars() (p. 3357)	QNX Neutrino (p. 46)	No	No	Yes	Yes
tcsendbreak() (p. 3362)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcsetattr() (p. 3364)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcsetpgrp() (p. 3367)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
tcsetsid() (p. 3369)	QNX Neutrino (p. 46)	No	No	No	Yes
tcsetsize() (p. 3371)	QNX Neutrino (p. 46)	No	No	Yes	Yes
tell() (p. 3373)	QNX 4 (p. 46)	Yes	No	Yes	Yes
tell64() (p. 3373)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
telldir() (p. 3375)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
tempnam() (p. 3377)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
thread_pool_control() (p. 3385)	QNX Neutrino (p. 46)	Yes	No	No	Yes

Function	Classification	Cancel	ISR	Signal	Thread
thread_pool_create() (p. 3387)	QNX Neutrino (p. 46)	No	No	No	Yes
thread_pool_destroy() (p. 3393)	QNX Neutrino (p. 46)	Yes	No	No	Yes
thread_pool_limits() (p. 3395)	QNX Neutrino (p. 46)	Yes	No	No	Yes
thread_pool_start() (p. 3397)	QNX Neutrino (p. 46)	Yes	No	No	Yes
ThreadCancel() (p. 3399)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadCancel_r() (p. 3399)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadCreate() (p. 3402)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadCreate_r() (p. 3402)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadCtl() (p. 3408)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadCtl_r() (p. 3408)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadDestroy() (p. 3417)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadDestroy_r() (p. 3417)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadDetach() (p. 3420)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadDetach_r() (p. 3420)	QNX Neutrino (p. 46)	No	No	Yes	Yes
ThreadJoin() (p. 3422)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
ThreadJoin_r() (p. 3422)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
time() (p. 3425)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
timer_create() (p. 3427)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
timer_delete() (p. 3431)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
timer_getexpstatus() (p. 3433)	QNX Neutrino (p. 46)	No	No	Yes	Yes
timer_getoverrun() (p. 3435)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
timer_gettime() (p. 3437)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
timer_settime() (p. 3439)	POSIX 1003.1 TMR (p. 46)	No	No	Yes	Yes
timer_timeout() (p. 3442)	QNX Neutrino (p. 46)	No	No	Yes	Yes
timer_timeout_r() (p. 3442)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerAlarm() (p. 3448)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerAlarm_r() (p. 3448)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerCreate() (p. 3451)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerCreate_r() (p. 3451)	QNX Neutrino (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
TimerDestroy() (p. 3454)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerDestroy_r() (p. 3454)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerInfo() (p. 3456)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerInfo_r() (p. 3456)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerSettime() (p. 3460)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerSettime_r() (p. 3460)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerTimeout() (p. 3464)	QNX Neutrino (p. 46)	No	No	Yes	Yes
TimerTimeout_r() (p. 3464)	QNX Neutrino (p. 46)	No	No	Yes	Yes
times() (p. 3471)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
timespec2nsec() (p. 3474)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
tmpfile() (p. 3478)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
tmpfile64() (p. 3478)	Large-file support (p. 46)	Yes	No	No	Yes
tmpnam() (p. 3480)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Read the Caveats
tolower() (p. 3482)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
toupper() (p. 3484)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
towctrans() (p. 3486)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
towlower() (p. 3488)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
towupper() (p. 3490)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
trace_func_enter() (p. 3492)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
trace_func_exit() (p. 3494)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
trace_here() (p. 3496)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
trace_logb() (p. 3498)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
trace_logbc() (p. 3500)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
trace_logf() (p. 3502)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
trace_logi() (p. 3504)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
trace_nlogf() (p. 3506)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
trace_vnlogf() (p. 3508)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
TraceEvent() (p. 3510)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
traceparser() (p. 3524)	QNX Neutrino (p. 46)	No	No	No	No
traceparser_cs() (p. 3526)	QNX Neutrino (p. 46)	No	No	No	No
traceparser_cs_range() (p. 3529)	QNX Neutrino (p. 46)	No	No	No	No
traceparser_debug() (p. 3531)	QNX Neutrino (p. 46)	No	No	No	Yes
traceparser_destroy() (p. 3533)	QNX Neutrino (p. 46)	No	No	No	No
traceparser_get_info() (p. 3534)	QNX Neutrino (p. 46)	No	No	No	No
traceparser_init() (p. 3537)	QNX Neutrino (p. 46)	No	No	No	No
truncate() (p. 3539)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
truncate64() (p. 3539)	Large-file support (p. 46)	No	No	Yes	Yes
ttyname() (p. 3542)	POSIX 1003.1 (p. 46)	No	No	No	No
ttyname_r() (p. 3544)	POSIX 1003.1 TSF (p. 46)	No	No	Yes	Yes
tzset() (p. 3547)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes

U

Function	Classification	Cancel	ISR	Signal	Thread
ualarm() (p. 3550)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
ultoa() (p. 3555)	QNX 4 (p. 46)	No	Yes	Yes	Yes
ulltoa() (p. 3555)	Unix (p. 46)	No	Yes	Yes	Yes
umask() (p. 3557)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
umount() (p. 3559)	QNX Neutrino (p. 46)	Yes	No	No	Yes
UNALIGNED_PUT16() (p. 3561)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
UNALIGNED_PUT32() (p. 3563)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
UNALIGNED_PUT64() (p. 3565)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
UNALIGNED_RET16() (p. 3567)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
UNALIGNED_RET32() (p. 3568)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
UNALIGNED_RET64() (p. 3569)	QNX Neutrino (p. 46)	No	Yes	Yes	Yes
uname() (p. 3571)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
ungetc() (p. 3573)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
ungetwc() (p. 3575)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
unlink() (p. 3579)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
unlockpt() (p. 3582)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
unsetenv() (p. 3584)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
usleep() (p. 3586)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
utime() (p. 3588)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
utimes() (p. 3591)	POSIX 1003.1 XSI (p. 46)	No	No	Yes	Yes
utmpname() (p. 3596)	Unix (p. 46)	Yes	No	Yes	Yes
utoa() (p. 3598)	QNX 4 (p. 46)	No	Yes	Yes	Yes

V

Function	Classification	Cancel	ISR	Signal	Thread
va_arg() (p. 3602)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
va_copy() (p. 3605)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
va_end() (p. 3607)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
va_start() (p. 3608)	POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
valloc() (p. 3610)	QNX Neutrino (p. 46)	No	No	Yes	Yes
verr() (p. 3612)	Unix (p. 46)	Yes	No	Yes	Yes
verrx() (p. 3612)	Unix (p. 46)	Yes	No	Yes	Yes
vfork() (p. 3614)	POSIX 1003.1 XSI (p. 46)	No	No	No	No
vfprintf() (p. 3616)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
vfscanf() (p. 3618)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
vfwprintf() (p. 3620)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
vfwscanf() (p. 3622)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
vprintf() (p. 3624)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
vscanf() (p. 3626)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes
vslog2f() (p. 3628)	QNX Neutrino (p. 46)	No	Read the Caveats	Yes	Yes
vslogf() (p. 3630)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
vsnprintf() (p. 3632)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
vsprintf() (p. 3635)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
vsscanf() (p. 3637)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes

Function	Classification	Cancel	ISR	Signal	Thread
vswprintf() (p. 3639)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
vswscanf() (p. 3641)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Read the Caveats	Yes
vsyslog() (p. 3643)	Unix (p. 46)	Yes	No	No	No
vwarn() (p. 3645)	Unix (p. 46)	Yes	No	Yes	Yes
vwarnx() (p. 3645)	Unix (p. 46)	Yes	No	Yes	Yes
vwprintf() (p. 3647)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
vwscanf() (p. 3649)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	No	Yes

W

Function	Classification	Cancel	ISR	Signal	Thread
wait() (p. 3652)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
wait3() (p. 3655)	Unix (p. 46)	Yes	No	Yes	Yes
wait4() (p. 3658)	Unix (p. 46)	Yes	No	Yes	Yes
waitfor() (p. 3661)	QNX Neutrino (p. 46)	No	No	No	Yes
waitid() (p. 3663)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
waitpid() (p. 3666)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
warn() (p. 3669)	Unix (p. 46)	Yes	No	Yes	Yes
warnx() (p. 3669)	Unix (p. 46)	Yes	No	Yes	Yes
wcr tomb() (p. 3671)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcscat() (p. 3673)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcschr() (p. 3674)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcscmp() (p. 3676)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcscoll() (p. 3678)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	No
wcscpy() (p. 3680)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcscspn() (p. 3682)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsftime() (p. 3684)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcslen() (p. 3686)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsncat() (p. 3687)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsncmp() (p. 3689)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsncpy() (p. 3691)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsspbrk() (p. 3693)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsrchr() (p. 3695)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsrtombs() (p. 3697)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcsspncpy() (p. 3699)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsstr() (p. 3701)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcstod() (p. 3703)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes

Function	Classification	Cancel	ISR	Signal	Thread
wcstof() (p. 3703)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcstold() (p. 3703)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcstoimax() (p. 3705)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcstoumax() (p. 3705)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcstok() (p. 3707)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcstol() (p. 3709)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcstoll() (p. 3709)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcstombs() (p. 3711)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wcstoul() (p. 3713)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcstoull() (p. 3713)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wcsxfrm() (p. 3715)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wctob() (p. 3717)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wctomb() (p. 3718)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wctrans() (p. 3720)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wctype() (p. 3722)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wmemchr() (p. 3724)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wmemcmp() (p. 3726)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wmemcpy() (p. 3728)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wmemmove() (p. 3730)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wmemset() (p. 3732)	ANSI (p. 46) POSIX 1003.1 (p. 46)	No	Yes	Yes	Yes
wordexp() (p. 3734)	POSIX 1003.1 (p. 46)	Yes	No	Yes	Yes
wordfree() (p. 3735)	POSIX 1003.1 (p. 46)	No	No	Yes	Yes
wprintf() (p. 3736)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes
write() (p. 3737)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
writeblock() (p. 3742)	QNX Neutrino (p. 46)	Yes	No	Yes	Yes
writev() (p. 3745)	POSIX 1003.1 XSI (p. 46)	Yes	No	Yes	Yes
wscanf() (p. 3748)	ANSI (p. 46) POSIX 1003.1 (p. 46)	Yes	No	No	Yes

Y

Function	Classification	Cancel	ISR	Signal	Thread
y0() (p. 3754)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
y0f() (p. 3754)	Unix (p. 46)	No	No	No	Yes
y1() (p. 3756)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
y1f() (p. 3756)	Unix (p. 46)	No	No	No	Yes
yn() (p. 3758)	POSIX 1003.1 XSI (p. 46)	No	No	No	Yes
ynf() (p. 3758)	Unix (p. 46)	No	No	No	Yes

Appendix C

What's New in this Reference?

This appendix describes the updates to this document in the following releases:

Release	New	Deprecated	Changed	Errata
QNX Neutrino 6.6	New	Deprecated	Changed	Errata
QNX Neutrino 6.5.0 Service Pack 1	New		Changed	Errata
QNX Neutrino 6.5.0	New		Changed	Errata
QNX Neutrino 6.4.1	New	Deprecated	Changed	Errata
QNX Neutrino 6.4.0	New	Deprecated	Changed	Errata
QNX Neutrino 6.3.2				Errata
QNX Neutrino Core OS 6.3.2	New		Changed	
QNX Neutrino 6.3.0 Service Pack 2	New		Changed	Errata
QNX Neutrino 6.3.0 Service Pack 1	New		Changed	Errata
QNX Neutrino 6.3.0	New	Deprecated		
QNX Neutrino 6.2.1	New		Changed	Errata
QNX Neutrino 6.2	New	Deprecated		Errata
QNX Neutrino 6.1.0	New	Deprecated		

What's new in QNX Neutrino 6.6?

New entries

[*acl_add_perm\(\)*](#) (p. 64)

Add a permission to an access control list (ACL) permission set

[*acl_calc_mask\(\)*](#) (p. 66)

Calculate the group class mask for an access control list (ACL)

[*acl_clear_perms\(\)*](#) (p. 68)

Clear all permissions from an ACL permission set

[*acl_copy_entry\(\)*](#) (p. 70)

Copy the contents of one ACL entry into another

[*acl_copy_ext\(\)*](#) (p. 72)

Copy an ACL from system space to user space

[*acl_copy_int\(\)*](#) (p. 74)

Copy an ACL from user space to system space

[*acl_create_entry\(\)*](#) (p. 76)

Create an entry in an access control list

[*acl_delete_entry\(\)*](#) (p. 78)

Delete an entry from an access control list (ACL)

[*acl_del_perm\(\)*](#) (p. 80)

Delete a permission from an ACL permissions set

[*acl_dup\(\)*](#) (p. 82)

Duplicate an access control list (ACL) in a working storage area

[*acl_free\(\)*](#) (p. 84)

Free the working storage area allocated for an access control list (ACL) data object

[*acl_from_text\(\)*](#) (p. 86)

Create an access control list (ACL) from text

[*acl_get_entry\(\)*](#) (p. 88)

Get an entry in an access control list (ACL)

[*acl_get_fd\(\)*](#) (p. 90)

Get the access control list associated with a file descriptor

[*acl_get_file\(\)*](#) (p. 92)

Get the ACL for a given path

[*acl_get_permset\(\)*](#) (p. 94)

Get a permission set from an ACL entry

[*acl_get_qualifier\(\)*](#) (p. 96)

Get the qualifier from an ACL entry

[*acl_get_tag_type\(\)*](#) (p. 98)

Get the type of tag from an ACL entry

[*acl_init\(\)*](#) (p. 100)

Allocate and initialize an access control list (ACL) working storage area

[*acl_set_fd\(\)*](#) (p. 102)

Set the access ACL for the object associated with a file descriptor

[*acl_set_file\(\)*](#) (p. 104)

Set the access control list for a path

[*acl_set_permset\(\)*](#) (p. 106)

Set the permissions set in an ACL entry

[*acl_set_qualifier\(\)*](#) (p. 108)

Set the qualifier for an ACL entry

[*acl_set_tag_type\(\)*](#) (p. 110)

Set the tag type of an ACL entry

[*acl_size\(\)*](#) (p. 112)

Determine the size of the external form of an ACL

[acl_to_text\(\)](#) (p. 114)

Convert an access control list (ACL) into text

[acl_valid\(\)](#) (p. 116)

Validate an access control list (ACL)

[aio_read64\(\)](#) (p. 131)

Asynchronously read from a file

[aio_write64\(\)](#) (p. 138)

Asynchronously write to a file

[aiocb64](#) (p. 141)

Asynchronous I/O control block

[ConnectClientInfoAble\(\)](#) (p. 360)

Store extended information about a client connection, including abilities

[ClientInfoExtFree\(\)](#) (p. 304)

Free the structure used for extended information about a client connection

[ConnectClientInfoExt\(\)](#) (p. 363)

Store extended information about a client connection

[dhcctl*\(\)](#) (p. 412)

Functions for controlling DHCP

[dirent64](#) (p. 422)

Data structure for a directory entry

[fdistrusted\(\)](#) (p. 633)

Determine if a file is trusted

[fs_crypto_check\(\)](#) (p. 727)

Determine if the underlying filesystem supports encryption

[fs_crypto_domain_add\(\)](#) (p. 729)

Create a domain

[fs_crypto_domain_key_change\(\)](#) (p. 732)

Change the key for a domain

[*fs_crypto_domain_key_check\(\)*](#) (p. 735)

Determine if a given domain key is valid

[*fs_crypto_domain_key_size\(\)*](#) (p. 738)

Return the size of keys used for filesystem encryption

[*fs_crypto_domain_lock\(\)*](#) (p. 739)

Lock a domain

[*fs_crypto_domain_query\(\)*](#) (p. 741)

Get status information for a domain

[*fs_crypto_domain_remove\(\)*](#) (p. 743)

Destroy a domain

[*fs_crypto_domain_unlock\(\)*](#) (p. 745)

Unlock a domain

[*fs_crypto_enable\(\)*](#) (p. 748)

Enable encryption support on a filesystem

[*fs_crypto_enable_option\(\)*](#) (p. 750)

Enable encryption support on a filesystem

[*fs_crypto_file_get_domain\(\)*](#) (p. 753)

Get the domain for a file or directory

[*fs_crypto_file_set_domain\(\)*](#) (p. 766)

Set the domain for a file or directory

[*fs_crypto_key_gen\(\)*](#) (p. 755)

Generate an encryption key from a password

[*fs_crypto_migrate_control\(\)*](#) (p. 757)

Control encryption migration within the filesystem

[*fs_crypto_migrate_path\(\)*](#) (p. 762)

Mark an entire directory for migration into an encrypted domain

***fs_crypto_migrate_status()* (p. 760)**

Get the status of migration in the filesystem

***fs_crypto_migrate_tag()* (p. 764)**

Mark a file for migration into an encrypted domain

***fs_crypto_set_logging()* (p. 769)**

Set the logging destination and verbosity.

***gettid()* (p. 981)**

Get the thread ID

***grantpt()* (p. 1009)**

Grant access to a slave pseudo-terminal device

***InterruptCharacteristic(), InterruptCharacteristic_r()* (p. 1159)**

Get or set a characteristic associated with an interrupt

***inotify_add_watch()* (p. 1135)**

Add or update a watch for filesystem events associated with a path

***inotify_event* (p. 1137)**

Structure that describes a watched filesystem event

***inotify_init()* (p. 1139)**

Initialize the inode notify system

***inotify_rm_watch()* (p. 1141)**

Remove the watch associated with the given watch descriptor

***iofunc_acl()* (p. 1212)**

Handle an IO_ACL message

***iofunc_acl_default()* (p. 1215)**

Default handler for the *acl* callout

***iofunc_client_info_able()* (p. 1244)**

Store information about a client connection, including abilities.

***iofunc_client_info_ext()* (p. 1246)**

Store information about a client connection. This function is an extended version of *iofunc_client_info()*—which is now deprecated—that can handle more than NGROUPS_MAX supplementary groups.

***iofunc_client_info_ext_free()* (p. 1248)**

Free the structure used for information about a client connection

***iofunc_devctl_verify()* (p. 1263)**

Perform validation that's common to all handlers for _IO_DEVCTL messages

***isblank()* (p. 1429)**

Test a character to see if it's a blank character

***iswblank()* (p. 1459)**

Test a wide character to see if it's a blank character

***hwi_next_item()* (p. 1032)**

Find the next item in the *hwinfo* structure

***hwi_next_tag()* (p. 1035)**

Find the next tag in the *hwinfo* structure

***lio_listio64()* (p. 1519)**

Initiate a list of I/O requests

***lldiv()* (p. 1508)**

Perform division on long long integers

***lockf64()* (p. 1535)**

Lock or unlock a section of a file

***memset_s()* (p. 1622)**

Securely set memory to a given value

***OMAPI* (p. 1902)**

Object Management Application Programming Interface

***posix_fadvise()* (p. 2022), *posix_fadvise64()* (p. 2022)**

Provide advisory information about the expected use of a file

***posix_fallocate()* (p. 2025), *posix_fallocate64()* (p. 2025)**

Allocate space for a file on the filesystem

[*posix_madvise\(\)*](#) (p. 2028)

Provide advisory information about the expected use of memory

[*posix_openpt\(\)*](#) (p. 2035)

Open a pseudo-terminal device

[*procmgr_ability\(\)*](#) (p. 2141)

Control a process's ability to perform certain operations

[*procmgr_timer_tolerance\(\)*](#) (p. 2178)

Get or set the default timer tolerance for a process

[*pthread_attr_getstack\(\)*](#) (p. 2203)

Get the thread-creation stack attributes

[*pthread_attr_setstack\(\)*](#) (p. 2227)

Set the thread-creation stack attributes

[*ptsname\(\)*](#) (p. 2436)

Get the name of a slave pseudo-terminal device

[*ptsname_r\(\)*](#) (p. 2438)

Get the name of a slave pseudo-terminal device (reentrant)

[*readdir64\(\)*](#) (p. 2514)

Read a directory entry

[*set_lowest_fd\(\)*](#) (p. 2843)

Specify the lowest acceptable file descriptor

[*slog2c\(\)*](#) (p. 3044)

Log a constant string in a slog2 buffer

[*slog2f\(\)*](#) (p. 3046)

Log a printf-style string in a slog2 buffer

[*slog2fa\(\)*](#) (p. 3048)

Log a printf-style string in a slog2 buffer

***slog2_get_verbosity()* (p. 3051)**

Get the verbosity level for the entire buffer set

***slog2_register()* (p. 3052)**

Register and allocate a new slog2 buffer set

***slog2_reset()* (p. 3056)**

Reset the state of sLogger2 for the process

***slog2_set_default_buffer()* (p. 3057)**

Set the default slog2 logging buffer

***slog2_set_verbosity()* (p. 3059)**

Set the verbosity level for the entire buffer set

***strlen()* (p. 3227)**

Compute the length of a string, to a maximum number of bytes

***sysmgr_runstate_burst()* (p. 3322)**

Ask the system to turn on dynamically offlined CPUs

***truncate64()* (p. 3539)**

Truncate a file to a specified length

***unlockpt()* (p. 3582)**

Unlock a pseudo-terminal master/slave pair

***vslog2f()* (p. 3628)**

Log a printf-style string in a slog2 buffer (varargs)

***waitfor()* (p. 3661)**

Wait for a path to appear

Deprecated content

We've deprecated our version of SNMP:

- *read_main_config_file()*
- *snmp_close()*
- *snmp_free_pdu()*
- *snmp_open()*

- *snmp_pdu_create()*
- *snmp_read()*
- *snmp_select_info()*
- *snmp_send()*
- *snmp_timeout()*

Use a third-party solution instead.

Changed content

ChannelCreate() (p. 278)

- There's a new `_NTO_CHF_PRIVATE` flag that you can use to mark the channel as being private.
- More than one channel in a process can now have one or both of `_NTO_CHF_COID_DISCONNECT` and `_NTO_CHF_THREAD_DEATH` set.

clock_settime() (p. 319), *ClockTime()* (p. 333)

The ability to set the system time is now controlled by the `PROCMGR_AID_CLOCKSET` ability, which you can set by calling *procmgr_ability()* (p. 2141).

ClockPeriod() (p. 329)

- The ability to change the clock period is now controlled by the `PROCMGR_AID_CLOCKPERIOD` ability, which you can set by calling *procmgr_ability()* (p. 2141).
- You can now call this function with a clock ID of `CLOCK_HARMONIC`, in order to set or get the harmonic timer boundary. Timers whose remaining time before expiry is greater than this boundary are considered for timer harmonization.
- This function now returns the period for all valid clock IDs.

confstr() (p. 344)

- The ability to set configuration strings is now controlled by the `PROCMGR_AID_CONFSET` ability, which you can set by calling *procmgr_ability()* (p. 2141).
- The documentation now states that if the value, including the terminating `NULL`, is greater than *len* bytes long, it's truncated to *len* - 1 bytes and terminated with a `NULL` character.

***ConnectAttach()* (p. 351)**

The ability to establish a connection between a process and a channel is now controlled by the `PROCmgr_AID_CONNECTION` ability, which you can set by calling *procmgr_ability()* (p. 2141).

***ConnectClientInfo()* (p. 356)**

We've described the *flags* member of the `_client_info` structure.

***ConnectFlags()* (p. 369)**

- There's a new `_NTO_COF_INSECURE` flag that the kernel sets if the client program replaces its process image (e.g., by calling *exec*()* (p. 487)) with a `setuid` or `setgid` binary. Until the process that owns the channel clears this flag, calls to *MsgDeliverEvent()* (p. 1736) fail with an error of `EACCES`. *ConnectFlags()* now fails with an error of `EINVAL` if the *mask* includes any invalid bits.
- The ability to change the connection flags on a coid for another process with a different user ID is now controlled by the `PROCmgr_AID_CONNECTION` ability, which you can set by calling *procmgr_ability()* (p. 2141).

***crypt()* (p. 385)**

We've deprecated the version of this function in `libc`. You should use the improved version that's in `liblogin`.

***devctl()* (p. 403)**

This function returns `ENXIO` if there's no such device or address (e.g., there's no media available to be loaded in the device represented by the file descriptor).

***dlopen()* (p. 454)**

The ability to load shared objects is now controlled by the `PROCmgr_AID_PROT_EXEC` ability, which you can set by calling *procmgr_ability()* (p. 2141).

***errno* (p. 539)**

- `EALREADY` is now defined as `EALREADY_NEW`, so that `EALREADY` and `EBUSY` have different values, as specified by POSIX. For more information, see “*Changes to EALREADY* (p. 547)” in the documentation for *errno*.

- We've added the numerical value for each error code, and indicated which constants are defined by POSIX.

[exec\(\)](#) (p. 550), [execle\(\)](#) (p. 556), [execlp\(\)](#) (p. 562), [execlpe\(\)](#) (p. 568), [execv\(\)](#) (p. 573), [execve\(\)](#) (p. 579), [execvp\(\)](#) (p. 585), [execvpe\(\)](#) (p. 591)

The ability to create child processes is now controlled by the PROC_MGR_AID_SPAWN ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[flock\(\)](#) (p. 666)

You no longer have to open a file with write permission in order to apply a lock on it.

[fork\(\)](#) (p. 685)

- The ability to fork processes is now controlled by the PROC_MGR_AID_FORK ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).
- You can now fork a multithreaded process. The child process has a single thread, regardless of the number that the parent process has.



If your program uses mutexes, you need to register a [pthread_atfork\(\)](#) handler that locks all the mutexes before you fork.

[fscanf\(\)](#) (p. 770), [fwscanf\(\)](#) (p. 810)

Note that these functions are generally considered to be unsafe for string handling. We've improved the examples by adding limits to the number of characters to be read.

[getaddrinfo\(\)](#) (p. 818)

This function is now thread-safe, as required by POSIX.

[getgrouplist\(\)](#) (p. 861)

This function no longer ignores the *basegid* argument, but automatically adds it to the list of group IDs.

[getgroups\(\)](#) (p. 864)

This function can now return up to `sysconf(_SC_NGROUPS_MAX)` group IDs. The limit used to be `NGROUPS_MAX`.

***getifaddrs()* (p. 885)**

This function can set *errno* to ENOMEM if the system is out of memory, or the interface list was growing while *getifaddrs()* was executing.

***getnameinfo()* (p. 894)**

This function is now thread-safe, as required by POSIX.

***getpgid()* (p. 917), *getsid()* (p. 958)**

The ability to get the group ID or session ID of a process outside the calling process's session is now controlled by the PROCMDR_AID_GETID ability, which you can set by calling *procmgr_ability()* (p. 2141).

***getsockopt()* (p. 962)**

You can now use the SO_BINDTODEVICE option for UDP, TCP, and RAW sockets.

***h_errno* (p. 1012)**

This variable is now thread-safe.

***initgroups()* (p. 1131)**

This function can now handle up to `sysconf(_SC_NGROUPS_MAX)` group IDs. The limit used to be NGROUPS_MAX.

***InterruptAttach()* (p. 1147)**

- The ability to attach interrupt handlers and events is now controlled by the PROCMDR_AID_INTERRUPT ability. Threads that call these functions (except for *InterruptHookIdle()*) also need I/O privileges, which are controlled by the PROCMDR_AID_IO ability. Use *procmgr_ability()* (p. 2141) to enable these abilities.
- We've listed where to look for interrupt vector numbers for your board.

***InterruptAttach()* (p. 1147), *InterruptAttachEvent()* (p. 1154), *InterruptHookIdle()* (p. 1168), *InterruptHookTrace()* (p. 1171)**

The ability to attach interrupt handlers and events is now controlled by the PROCMDR_AID_INTERRUPT ability. Threads that call these functions (except for *InterruptHookIdle()*) also need I/O privileges, which are controlled by the PROCMDR_AID_IO ability. Use *procmgr_ability()* (p. 2141) to enable these abilities.

***iofunc_attr_init()* (p. 1217)**

In general, there isn't a “correct” set of permissions to use for the resource; you should restrict them according to what you want other processes and users to be able to do with your resource manager.

[*iofunc_och_alloc\(\)*](#) (p. 1313)

This function sets *errno* to ENFILE if it can't allocate the OCB. If you override this function in your resource manager, your replacement function should do the same.

[*iofunc_mmap\(\)*](#) (p. 1294)

The *io_mmap_t* structure now includes a new input field, *required_prot*, and a new output field, *allowed_prot*.

[*iofunc_stat\(\)*](#) (p. 1353), [*iofunc_stat_default\(\)*](#) (p. 1355)

These functions now return an *errno* (p. 539) value if an error occurred.

[*iofunc_write_verify\(\)*](#) (p. 1382)

This function now returns EBADMSG if the length of the message is invalid.

[*isalnum\(\)*](#) (p. 1421), [*isalpha\(\)*](#) (p. 1423), [*isblank\(\)*](#) (p. 1429), [*iscntrl\(\)*](#) (p. 1431), [*isdigit\(\)*](#) (p. 1433), [*isgraph\(\)*](#) (p. 1439), [*islower\(\)*](#) (p. 1443), [*isprint\(\)*](#) (p. 1447), [*ispunct\(\)*](#) (p. 1449), [*isspace\(\)*](#) (p. 1451), [*isupper\(\)*](#) (p. 1453), [*isxdigit\(\)*](#) (p. 1481)

Note that these functions take an integer argument, but the value must be representable as an unsigned char or be EOF; the behavior for other values is undefined.

[*iswalnum\(\)*](#) (p. 1455), [*iswalpha\(\)*](#) (p. 1457), [*iswblank\(\)*](#) (p. 1459), [*iswcntrl\(\)*](#) (p. 1461), [*iswdigit\(\)*](#) (p. 1465), [*iswgraph\(\)*](#) (p. 1467), [*iswlower\(\)*](#) (p. 1469), [*iswprint\(\)*](#) (p. 1471), [*iswpunct\(\)*](#) (p. 1473), [*iswspace\(\)*](#) (p. 1475), [*iswupper\(\)*](#) (p. 1477), [*iswxdigit\(\)*](#) (p. 1479)

Note that these functions take a *wint_t* argument, but the value must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; the behavior for other values is undefined.

[*kill\(\)*](#) (p. 1494)

The ability to send signals is now controlled by the PROCmgr_AID_SIGNAL ability, which you can set by calling [*procmgr_ability\(\)*](#) (p. 2141).

[*malloc\(\)*](#) (p. 1569)

- The heap is protected by such security measures as address space layout randomization (ASLR) and heap cookies.

- The **MALLOC_MEMORY_BANDCONFIG** environment variable is now ignored.

[malloc\(\)](#) (p. 1575)

We've replaced libmalloc with librcheck.

[memalign\(\)](#) (p. 1603)

This function now indicates an error of EINVAL if the given alignment is greater than the value of the **MALLOC_MAX_ALIGNMENT** environment variable.

[memcpy\(\)](#) (p. 1605)

In order to conform to POSIX, this function now converts its *c* argument into an unsigned char.

[memcpyv\(\)](#) (p. 1613)

We've improved the example.

[mlock\(\)](#) (p. 1651), [mlockall\(\)](#) (p. 1653)

The ability to lock memory is now controlled by the PROCmgr_AID_MEM_LOCK ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[mmap\(\)](#) (p. 1656)

- The ability to do certain operations is now controlled by [procmgr_ability\(\)](#) (p. 2141):

The ability to:	Is controlled by:
Map fixed addresses	PROCmgr_AID_MAP_FIXED
Map physical memory	PROCmgr_AID_MEM_PHYS
Map memory as executable	PROCmgr_AID_PROT_EXEC

- This function gives an error of EACCES if you specify PROT_EXEC for a memory-mapped file mapping, the file doesn't have execute permission for the client process, and procmgr was started with the -mX option.

[mmap_device_io\(\)](#) (p. 1666), [mmap_device_memory\(\)](#) (p. 1668)

In order to map physical memory, your process needs to have the PROCmgr_AID_MEM_PHYS ability enabled. See [procmgr_ability\(\)](#) (p. 2141).

[mprotect\(\)](#) (p. 1699)

This function gives an error of `EACCES` if you specify `PROT_EXEC` for a memory-mapped file mapping, the file doesn't have execute permission for the client process, and `procnto` was started with the `-mX` option.

[*MsgDeliverEvent\(\)*](#) (p. 1736)

- As a convenience, you can specify a *rcvid* of 0 to target the event at the active thread.
- If the client program replaces its process image (e.g., by calling [*exec*\(\)*](#) (p. 487)) with a `setuid` or `setgid` binary, the kernel now sets the `_NTO_COF_INSECURE` flag on the channel. Until the process that owns the channel uses [*ConnectFlags\(\)*](#) (p. 369) to clear this flag, calls to *MsgDeliverEvent()* fail with an error of `EACCES`.

[*_msg_info*](#) (p. 1731)

There's a new `_NTO_MI_CONSTRAINED` flag that indicates that the message was received from a resource-constrained client.

[*MsgKeyData\(\)*](#) (p. 1747)

- The ability to pass the data is now controlled by the `PROCmgr_AID_KEYDATA` ability, which you can set by calling [*procmgr_ability\(\)*](#) (p. 2141).
- Depending on your system's security requirements, you can use as a key a statistically random number returned by *rand()* or a value read from the more secure random number generator, `/dev/random`.

[*MsgReply\(\)*](#) (p. 1775), [*MsgReplyv\(\)*](#) (p. 1778)

These functions indicate an error of `ENOREMOTE` if the reply has to go across the network, and `Qnet` isn't running.

[*open\(\)*](#) (p. 1905)

The supported flags now include:

- `O_NOFOLLOW` — if the last component in the path is a symlink, make the open fail with an error of `ELOOP`.
- `O_NOSYMLINK` — if any component in the path is a symlink, make the open fail with an error of `ELOOP`.

Note that the `O_ASYNC` flag isn't currently supported.

[*pathmgr_symlink\(\)*](#) (p. 1943)

- The ability to create the symbolic link is now controlled by the `PROC_MGR_AID_PATHSPACE` ability, which you can set by calling [proc_mgr_ability\(\)](#) (p. 2141).
- Note that you can get some surprising results if you create a symbolic link to a directory that's a symbolic link to somewhere else. For more information, see “Symbolic links” in the Working with Filesystems chapter of the QNX Neutrino *User's Guide*.

[poll\(\)](#) (p. 2013)

If the remote end of a socket is closed, `poll()` now indicates both `POLLIN` and `POLLHUP` events.

[posix_memalign\(\)](#) (p. 2033)

This function now indicates an error of `EINVAL` if the given alignment is greater than the value of the `MALLOC_MAX_ALIGNMENT` environment variable.

[posix_spawn\(\)](#) (p. 2037), [posix_spawnnp\(\)](#) (p. 2037)

- The ability to do certain operations is controlled by [proc_mgr_ability\(\)](#) (p. 2141):

The ability to:	Is controlled by:
Spawn new processes	<code>PROC_MGR_AID_SPAWN</code>
Set the group ID of the child process	<code>PROC_MGR_AID_SPAWN_SETGID</code>
Set the user ID of the child process	<code>PROC_MGR_AID_SPAWN_SETUID</code>
Create a new application ID for the child process	<code>PROC_MGR_AID_CHILD_NEWAPP</code>

- The following flags are new:
 - `POSIX_SPAWN_SETSID`
 - `POSIX_SPAWN_DEBUG`
 - `POSIX_SPAWN_EXEC`
 - `POSIX_SPAWN_NEWAPP`
 - `POSIX_SPAWN_SEARCH_PATH`
 - `POSIX_SPAWN_CHECK_SCRIPT`
 - `POSIX_SPAWN_PADDR64_SAFE`
 - `POSIX_SPAWN_TCSETPGROUP`

- Note that in the QNX Neutrino implementation, fork handlers aren't run when you call these functions.

[*procmgr_event_notify\(\)*](#) (p. 2158), [*procmgr_event_trigger\(\)*](#) (p. 2170)

- The ability to trigger privileged events is now controlled by the PROCmgr_AID_EVENT ability, which you can set by calling [*procmgr_ability\(\)*](#) (p. 2141).
- There's a new PROCmgr_EVENT_TOD event that's triggered when a process changed the time of day.

[*procmgr_session\(\)*](#) (p. 2175)

- The ability to use the PROCmgr_SESSION_SETSID event is now controlled by the PROCmgr_AID_PGRP ability; the ability to use the PROCmgr_SESSION_SETPGRP, PROCmgr_SESSION_SIGNAL_PID, PROCmgr_SESSION_SIGNAL_PGRP, and PROCmgr_SESSION_SIGNAL_LEADER events is now controlled by the PROCmgr_AID_SESSION ability. You can set these abilities by calling [*procmgr_ability\(\)*](#) (p. 2141).
- The node descriptor that you pass to this function must currently be ND_LOCAL_NODE, or you'll get an error of ENOREMOTE.

[*pthread_cond_timedwait\(\)*](#) (p. 2265), [*pthread_cond_wait\(\)*](#) (p. 2269)

This functions give an error of EINVAL if the mutex has died.

[*pthread_getschedparam\(\)*](#) (p. 2299)

In order to get the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, the calling process must now have the PROCmgr_AID_SCHEDULE ability enabled. See [*procmgr_ability\(\)*](#) (p. 2141).

[*pthread_kill\(\)*](#) (p. 2310)

The ability to send a signal to a thread is now controlled by the PROCmgr_AID_SIGNAL ability, which you can set by calling [*procmgr_ability\(\)*](#) (p. 2141).

[*pthread_mutex_lock\(\)*](#) (p. 2318)

- This function gives an error of EINVAL if the mutex has died.
- The sample code now checks for errors.

[*pthread_mutex_setprioceiling\(\)*](#) (p. 2322), [*pthread_mutexattr_setprioceiling\(\)*](#) (p. 2349),
[*pthread_mutexattr_setprotocol\(\)*](#) (p. 2351)

The ability to set a mutex's priority ceiling to a value that's above the maximum priority for an unprivileged process is now controlled by the PROCmgr_AID_PRIORITY ability, which you can set by calling [*procmgr_ability\(\)*](#) (p. 2141).

[*pthread_mutex_timedlock\(\)*](#) (p. 2324), [*pthread_mutex_timedlock_monotonic\(\)*](#) (p. 2324),
[*pthread_mutex_trylock\(\)*](#) (p. 2327)

These functions give an error of EINVAL if the mutex has died.

[*pthread_setschedparam\(\)*](#) (p. 2400), [*pthread_setschedprio\(\)*](#) (p. 2402)

- In order to set its priority to a value above the maximum allowed for an unprivileged process, a process must now have the PROCmgr_AID_PRIORITY ability enabled. In order to set the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, the calling process must now have the PROCmgr_AID_SCHEDULE ability enabled. See [*procmgr_ability\(\)*](#) (p. 2141).
- You can wrap the new priority in one of these macros to specify how to handle out-of-range priority requests:
 - *SCHED_PRIO_LIMIT_ERROR(prio)* — indicate an error
 - *SCHED_PRIO_LIMIT_SATURATE(prio)* — saturate at the maximum allowed priority

If *procnto* was started with a -P option ending with s or S, out-of-range priority requests by default saturate at the maximum allowed value.

[*pthread_sigmask\(\)*](#) (p. 2407)

The ability to change the signal mask is now controlled by the PROCmgr_AID_SIGNAL ability, which you can set by calling [*procmgr_ability\(\)*](#) (p. 2141).

[*readdir_r\(\)*](#) (p. 2518)

Some filesystems support names that are longer than the value of NAME_MAX. You can use [*pathconf\(\)*](#) (p. 1936) with _PC_NAME_MAX to determine the maximum number of bytes (not including the terminating null) allowed in a file name for a particular filesystem.

[*resmgr_attach\(\)*](#) (p. 2580)

- The *flags* field in the *resmgr_attr_t* structure has a new flag, *RESMGR_FLAG_RCM*, that makes the resource manager automatically adopt the client's resource constraint mode when handling a request.
- The ability to add names to the pathname space is now controlled by the *PROCmgr_AID_PATHSPACE* ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).
- Note that if multiple registrations are made with *_RESMGR_FLAG_BEFORE* or *_RESMGR_FLAG_AFTER*, then all *BEFORE* registrations are resolved first in FIFO order, followed by the non-flagged registrations in LIFO order, and then all *AFTER* registrations in LIFO order.

[rsrddbmgr_attach\(\)](#) (p. 2674), [rsrddbmgr_detach\(\)](#) (p. 2686)

The ability to reserve a system resource for a process or return one to the resource database is now controlled by the *PROCmgr_AID_RSRCDBMGR* ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[scanf\(\)](#) (p. 2713)

Note that this function is generally considered to be unsafe for string handling. We've improved the example by adding limits to the number of characters to be read.

[SchedCtl\(\)](#) (p. 2745)

- In order to perform certain privileged operations, your process must now either be running as root or have the *PROCmgr_AID_APS_ROOT* ability enabled (see [procmgr_ability\(\)](#) (p. 2141)).
- The *SCHED_APS_SCHEDPOL_BMP_SAFETY* scheduling policy has been discontinued. If you try to choose this policy, *SchedCtl()* now indicates an error of *EINVAL*.

[SchedGet\(\)](#) (p. 2774), [SchedSet\(\)](#) (p. 2780)

In order to get or set the scheduling policy for a process whose real or saved user ID is different from the calling process's real or effective user ID, the calling process must now have the *PROCmgr_AID_SCHEDULE* ability enabled. See [procmgr_ability\(\)](#) (p. 2141).

[sched_param](#) (p. 2731)

When you're setting the priority, you can wrap the new priority in one these macros to specify how to handle out-of-range priority requests:

- *SCHED_PRIO_LIMIT_ERROR(p)* — indicate an error
- *SCHED_PRIO_LIMIT_SATURATE(p)* — saturate at the maximum allowed priority

If *procnto* was started with a *-P* option ending with *s* or *S*, out-of-range priority requests by default saturate at the maximum allowed value.

[*sched_setparam\(\)* \(p. 2738\)](#), [*sched_setscheduler\(\)* \(p. 2740\)](#), [*SchedSet\(\)* \(p. 2780\)](#), [*setprio\(\)* \(p. 2890\)](#)

In order to set its priority to a value above the maximum allowed for an unprivileged process, a process must now have the *PROCmgr_AID_PRIORITY* ability enabled. See [*procmgr_ability\(\)* \(p. 2141\)](#).

[*setgid\(\)* \(p. 2851\)](#), [*setgid\(\)* \(p. 2858\)](#), [*setregid\(\)* \(p. 2895\)](#), [*setgroups\(\)* \(p. 2864\)](#)

The ability to set a process's group ID, effective group ID, real and effective group IDs, or supplementary group IDs is now controlled by the *PROCmgr_AID_SETGID* ability, which you can set by calling [*procmgr_ability\(\)* \(p. 2141\)](#).

[*setgroups\(\)* \(p. 2864\)](#)

This function can now handle up to *sysconf(_SC_NGROUPS_MAX)* group IDs. The limit used to be *NGROUPS_MAX*.

[*seteuid\(\)* \(p. 2856\)](#), [*setreuid\(\)* \(p. 2897\)](#), [*setuid\(\)* \(p. 2917\)](#)

The ability to set a process's user ID, effective user ID, or both is now controlled by the *PROCmgr_AID_SETUID* ability, which you can set by calling [*procmgr_ability\(\)* \(p. 2141\)](#).

[*setpgrp\(\)* \(p. 2889\)](#)

The ability to is now controlled by the *PROCmgr_AID_PGRP* ability, which you can set by calling [*procmgr_ability\(\)* \(p. 2141\)](#). This ability is enabled by default (for POSIX conformance).

[*setprio\(\)* \(p. 2890\)](#)

You can wrap the new priority in one these macros to specify how to handle out-of-range priority requests:

- *SCHED_PRIO_LIMIT_ERROR(prio)* — indicate an error
- *SCHED_PRIO_LIMIT_SATURATE(prio)* — saturate at the maximum allowed priority

If *procnto* was started with a *-P* option ending with *s* or *S*, out-of-range priority requests by default saturate at the maximum allowed value.

[*setrlimit\(\)*](#) (p. 2899)

- In order to raise the value of a hard limit, the process must now have the PROC_MGR_AID_RLIMIT ability enabled. See [*proc_mgr_ability\(\)*](#) (p. 2141).
- There's a new RLIMIT_FREEMEM resource that you can use to limit the total memory usage.

[*setsid\(\)*](#) (p. 2907)

The ability to create new sessions is now controlled by the PROC_MGR_AID_PGRP ability, which you can set by calling [*proc_mgr_ability\(\)*](#) (p. 2141).

[*shm_ctl\(\)*](#) (p. 2925), [*shm_ctl_special\(\)*](#) (p. 2931)

- The ability to use the SHMCTL_GLOBAL flag is now controlled by the PROC_MGR_AID_MEM_GLOBAL ability, which you can set by calling [*proc_mgr_ability\(\)*](#) (p. 2141).
- There's a new SHMCTL_NODEFRAG flag that you can use with SHMCTL_PHYS | SHMCTL_ANON to specify that you want physically contiguous memory, but not so much that you're willing to wait for defragmentation. If contiguous memory isn't immediately available, the call gives an error of ENOMEM, and you can retry with just SHMCTL_ANON if you're willing to accept discontiguity.
- There's a new SHMCTL_REPEAT flag that you can use to create a strided physical object.
- These functions set *errno* to more explanatory values.
- On x86 platforms, SHMCTL_LAZYWRITE now causes Write Combining to be used.

[*sigaction\(\)*](#) (p. 2948), [*signal\(\)*](#) (p. 2974)

The ability to specify the action for a signal is now controlled by the PROC_MGR_AID_SIGNAL ability, which you can set by calling [*proc_mgr_ability\(\)*](#) (p. 2141).

[*SignalAction\(\)*](#) (p. 2977)

- The ability to specify the action for a signal is now controlled by the PROC_MGR_AID_SIGNAL ability, which you can set by calling [*proc_mgr_ability\(\)*](#) (p. 2141).
- Note that the *sigstubs* argument can be NULL if *act* is also NULL.
- Note that SIGDEADLK and SIGEMT refer to the same signal.

SignalKill() (p. 2986)

- The ability to send signals is now controlled by the PROC_MGR_AID_SIGNAL ability, which you can set by calling *procmgr_ability()* (p. 2141).
- There's a new SIG_APPID flag that you can OR into the *signo* argument. This tells *SignalKill()* to send the signal to all the processes with an application ID that matches the *pid* argument.
- If the node descriptor is for a remote node, but the network manager isn't running, you get an error of ENOREMOTE.

SignalProcmask() (p. 2991), ***sigprocmask()*** (p. 3004)

The ability to change the signal mask is now controlled by the PROC_MGR_AID_SIGNAL ability, which you can set by calling *procmgr_ability()* (p. 2141).

sigqueue() (p. 3007)

The ability to send signals is now controlled by the PROC_MGR_AID_SIGNAL ability, which you can set by calling *procmgr_ability()* (p. 2141).

socketpair() (p. 3078)

We've made the example clearer.

spawn() (p. 3089), ***spawnl()*** (p. 3099), ***spawnle()*** (p. 3105), ***spawnlp()*** (p. 3111), ***spawnlpe()*** (p. 3116), ***spawnnp()*** (p. 3122), ***spawnvp()*** (p. 3129), ***spawnve()*** (p. 3134), ***spawnvpe()*** (p. 3140), ***spawnvpe()*** (p. 3145)

- The ability to create child processes is now controlled by the PROC_MGR_AID_SPAWN ability, and the ability to create a new application ID for the child process is controlled by the PROC_MGR_AID_CHILD_NEWAPP ability. You can set these abilities by calling *procmgr_ability()* (p. 2141).
- There's a new SPAWN_NEWAPP flag that makes *spawn*()* assign a new application ID to the process. If you don't set this flag, the process inherits the parent's application ID.

strtok_r() (p. 3260)

Note that you might want to keep a copy of the original string because *strtok_r()* is likely to modify it.

SyncCtl() (p. 3286)

The ability to attach a pulse to a mutex is now controlled by the PROCmgr_AID_CONNECTION ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[SyncMutexEvent\(\)](#) (p. 3291)

- The ability to attach a pulse to a mutex is now controlled by the PROCmgr_AID_CONNECTION ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).
- The documentation now explains that the event is delivered when a mutex moves to the DEAD state. There's also a description of how you might manage the death of a process that has a mutex locked.

[SyncTypeCreate\(\)](#), [SyncTypeCreate_r\(\)](#) (p. 3305)

These functions give an error of EBUSY if the given synchronization object was previously initialized but hasn't been destroyed.

[sysmgr_reboot\(\)](#) (p. 3319)

The ability to reboot your computer is now controlled by the PROCmgr_AID_REBOOT ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[SYSPAGE_ENTRY\(\)](#) (p. 3326)

You can use this function to query CPU-specific information, such as whether or not the processor supports NEON.

[tcsetpgrp\(\)](#) (p. 3367)

The ability to set the process group ID is now controlled by the PROCmgr_AID_SESSION ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[ThreadCreate\(\)](#) (p. 3402)

We've documented the `__guardsize` and `__prealloc` members of the `_thread_attr` structure.

[ThreadCtl\(\)](#) (p. 3408)

- There's a new `_NTO_TCTL_IO_PRIV` command that's similar to `_NTO_TCTL_IO` but also requests that the thread be put into System (privileged) execution mode on ARM targets.
- There's a new command, `_NTO_TCTL_RCM_GET_AND_SET`, that you can use to enter or leave resource-constrained mode.

- On ARMv7 targets, the `_NTO_TCTL_ALIGN_FAULT` command is now ignored; you can no longer control the alignment-fault behavior on a per-thread basis on these targets. You can set the global behavior by using the `-ad` or `-ae` option to `procnto*`.
- The ability to superlock the process's memory and request I/O privileges by using `PROCmgr_AID_IO` is now controlled by the `PROCmgr_AID_IO` ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[timer_create\(\)](#) (p. 3427), [TimerCreate\(\)](#) (p. 3451), [timer_getexpstatus\(\)](#) (p. 3433), [timer_getoverrun\(\)](#) (p. 3435), [timer_gettime\(\)](#) (p. 3437), [TimerInfo\(\)](#) (p. 3456)

The ability to create a timer that sends a pulse is now controlled by the `PROCmgr_AID_TIMER` ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[TimerInfo\(\)](#) (p. 3456)

- The ability to get timer information is now controlled by the `PROCmgr_AID_TIMER` ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).
- The *flags* argument has a new bit: `_NTO_TI_REPORT_TOLERANCE`. If you specify it, the function stores the timer tolerance in the *otime.interval_nsec* member of the `_timer_info` structure.
- If the timer is a precise timer, `_NTO_TI_PRECISE` is set in the *flags* member of the `_timer_info` structure.

[timer_settime\(\)](#) (p. 3439), [TimerSettime\(\)](#) (p. 3460)

The following flags are new (as QNX Neutrino extensions for *timer_settime()*):

- `TIMER_PRECISE` — exclude the timer from timer harmonization.
- `TIMER_TOLERANCE` — specify the amount of the tolerance to allow the kernel in low-power situations.

[TimerTimeout\(\)](#) (p. 3464), [timer_timeout\(\)](#) (p. 3442)

You can OR `TIMER_TOLERANCE` into the *flags* argument if you want to specify the amount of the tolerance to allow the kernel in low-power situations. You can't use `TIMER_TOLERANCE` and `TIMER_ABSTIME` in the same call.

If you're setting an expiration time (i.e., you didn't specify `TIMER_TOLERANCE`), you can OR in `TIMER_PRECISE` to exclude the timer from timer harmonization.

[tm](#) (p. 3476)

Note that the *tm_zone* pointer in this structure could become stale or invalid after the next call to [tzset\(\)](#) (p. 3547).

[tolower\(\)](#) (p. 3482), [toupper\(\)](#) (p. 3484)

Note that these functions take an `integer` argument, but the value must be representable as an `unsigned char` or be EOF; the behavior for other values is undefined.

[tolower\(\)](#) (p. 3488), [toupper\(\)](#) (p. 3490)

Note that these functions take a `wint_t` argument, but the value must be a wide-character code that corresponds to a valid character in the current locale, or must be WEOF; the behavior for other values is undefined.

[TraceEvent\(\)](#) (p. 3510)

- The ability to add handlers for trace events or allocate the instrumented kernel's trace buffers is now controlled by the `PROCmgr_AID_TRACE` ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).
- There's a new `_NTO_TRACE_SETBUFFER` command that you can use to specify some previously allocated physical memory to use as the trace buffer. The intended application is to place the buffer in an area of memory that will be preserved across system resets, facilitating post-crash analysis.

[umask\(\)](#) (p. 3557)

The ability to change the file-mode creation mask for a process with a different effective user ID is now controlled by the `PROCmgr_AID_UMASK` ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[UNIX-domain protocol family](#) (p. 3577)

We now support abstract Unix-domain sockets, which are indicated by an address whose leading byte is `'\0'`. They don't create nodes in the path space, which in turn means they don't need to be unlinked.

[vfscanf\(\)](#) (p. 3618), [vfwscanf\(\)](#) (p. 3622), [vscanf\(\)](#) (p. 3626), [vwscanf\(\)](#) (p. 3649)

Note that these functions are generally considered to be unsafe for string handling. We've improved the examples by adding limits to the number of characters to be read.

[wait\(\)](#) (p. 3652), [wait3\(\)](#) (p. 3655), [wait4\(\)](#) (p. 3658), [waitid\(\)](#) (p. 3663), [waitpid\(\)](#) (p. 3666)

The ability to wait for the status of a terminated child process whose real or saved user ID is different from the calling process's real or effective user ID is now controlled by the `PROC_MGR_AID_WAIT` ability, which you can set by calling [procmgr_ability\(\)](#) (p. 2141).

[wscanf\(\)](#) (p. 3748)

Note that this function is generally considered to be unsafe for string handling.

Errata

[abs\(\)](#) (p. 56)

The behavior of this function is undefined if the result can't be represented as an `int`.

[bt_sprnf_addr\(\)](#) (p. 231)

This function returns -1 and sets `errno` if an error occurred.

Cancellation points

POSIX specifies a list of functions that *must* have cancellation points that occur when a thread is executing, and another of functions that *may* have cancellation points. No other function defined by POSIX can have cancellation points.

The following functions must have cancellation points but weren't marked as having them in the documentation:

- [fdatasync\(\)](#) (p. 631)
- [lockf\(\)](#) (p. 1535)
- [select\(\)](#) (p. 2791)

The following functions are defined by POSIX, are not permitted by POSIX to be cancellation points, but were marked as such in the documentation:

- [fchown\(\)](#) (p. 615)
- [fdopen\(\)](#) (p. 635)
- [fwide\(\)](#) (p. 804)
- [getpeername\(\)](#) (p. 915)
- [getpwent_r\(\)](#) (p. 932)
- [getsockname\(\)](#) (p. 960)
- [getsockopt\(\)](#) (p. 962)
- [if_indextoname\(\)](#) (p. 1050)
- [if_nameindex\(\)](#) (p. 1052)

- [*if_nametoindex\(\)*](#) (p. 1054)
- [*initstate\(\)*](#) (p. 1133)
- [*listen\(\)*](#) (p. 1523)
- [*mknod\(\)*](#) (p. 1642)
- [*mktemp\(\)*](#) (p. 1647)
- [*pthread_rwlock_tryrdlock\(\)*](#) (p. 2375)
- [*pthread_rwlock_trywrlock\(\)*](#) (p. 2377)
- [*realpath\(\)*](#) (p. 2531)
- [*setsockopt\(\)*](#) (p. 2909)
- [*shutdown\(\)*](#) (p. 2942)
- [*socketatmark\(\)*](#) (p. 3073)
- [*socket\(\)*](#) (p. 3075)
- [*socketpair\(\)*](#) (p. 3078)
- [*telldir\(\)*](#) (p. 3375)
- [*truncate\(\)*](#) (p. 3539)
- [*vfscanf\(\)*](#) (p. 3618)
- [*vwscanf\(\)*](#) (p. 3622)
- [*vscanf\(\)*](#) (p. 3626)
- [*vwscanf\(\)*](#) (p. 3649)

The following functions have cancellation points:

- [*fgetpos\(\)*](#) (p. 650)
- [*fseek\(\)*](#) (p. 772)
- [*fseeko\(\)*](#) (p. 772)
- [*fseeko64\(\)*](#) (p. 772)
- [*fsetpos\(\)*](#) (p. 776)
- [*ioctl\(\)*](#) (p. 1198)
- [*ioctl_socket\(\)*](#) (p. 1198)
- [*lseek\(\)*](#) (p. 1555)
- [*lseek64\(\)*](#) (p. 1555)
- [*pathconf\(\)*](#) (p. 1936)
- [*posix_spawn\(\)*](#) (p. 2037)
- [*posix_spawnnp\(\)*](#) (p. 2037)
- [*readdir_r\(\)*](#) (p. 2518)
- [*sethostent\(\)*](#) (p. 2866)
- [*vfprintf\(\)*](#) (p. 3616)
- [*vwprintf\(\)*](#) (p. 3620)
- [*vprintf\(\)*](#) (p. 3624)
- [*vwprintf\(\)*](#) (p. 3647)

***ConnectDetach()* (p. 367)**

We've clarified what happens if client threads are SEND- or REPLY-blocked on the connection being detached.

***devctl()* (p. 403)**

This function returns ENOTTY, not ENOSYS, if the device doesn't support the given command.

***dup2()* (p. 485)**

- We've corrected the list of error codes.
- This call:

```
dup_filedes = dup2( filedes, filedes2 );
```

isn't necessarily the same as:

```
close( filedes2 );
dup_filedes = fcntl( filedes, F_DUPFD, filedes2 );
```

***InterruptStatus()* (p. 1178)**

We've corrected the synopsis.

***iofunc_open_default()* (p. 1325)**

This function doesn't call *resmgr_open_bind()* directly; it calls *iofunc_ocb_attach()*, which calls *resmgr_open_bind()* in turn.

***mbtowc()* (p. 1595)**

This function isn't safe to use in multithreaded programs; use *mbrtowc()* (p. 1587) instead.

***MsgDeliverEvent()* (p. 1736)**

We've corrected the description of the circumstances that could cause an error of ESRVRFAULT.

***MsgRead()* (p. 1754), *MsgReadv()* (p. 1757), *MsgReceive()* (p. 1760), *MsgReceivePulse()* (p. 1765), *MsgReceivePulsev()* (p. 1768), *MsgReceivev()* (p. 1771), *MsgWrite()* (p. 1819), *MsgWritev()* (p. 1822)**

We've corrected the description of the circumstances that could cause an error of EFAULT.

[*MsgSend\(\)* \(p. 1781\)](#), [*MsgSendnc\(\)* \(p. 1785\)](#), [*MsgSendPulse\(\)* \(p. 1789\)](#), [*MsgSendsv\(\)* \(p. 1793\)](#),
[*MsgSendsvnc\(\)* \(p. 1797\)](#), [*MsgSendv\(\)* \(p. 1801\)](#), [*MsgSendvnc\(\)* \(p. 1805\)](#), [*MsgSendvs\(\)* \(p. 1809\)](#),
[*MsgSendvsnc\(\)* \(p. 1813\)](#)

We've corrected the description of the circumstances that could cause an error of ESRVRFault.

[*munmap_flags\(\)* \(p. 1838\)](#)

We've corrected the names of the UNMAP_INIT_REQUIRED and UNMAP_INIT_OPTIONAL flags.

[*name_attach\(\)* \(p. 1842\)](#), [*name_open\(\)* \(p. 1852\)](#)

A server that calls *name_attach()* receives an _IO_CONNECT message when a client calls *name_open()* only if the global name service, gns, is running.

[*poll\(\)* \(p. 2013\)](#)

Not all managers support POLLWRBAND.

[*posix_mem_offset\(\)* \(p. 2030\)](#)

We've corrected the description of this function.

[*printf\(\)* \(p. 2133\)](#)

It isn't true that if you give a precision specifier of *, but there's no precision value in the argument list, a precision of 0 is used. The compiler gives you a warning if you don't provide the right number of arguments or if the argument types don't match those indicated by the format string.

[*pthread_cond_destroy\(\)* \(p. 2259\)](#)

If another thread is blocked on the condition variable, this function doesn't return EBUSY; instead, that thread becomes unblocked.

[*realpath\(\)* \(p. 2531\)](#)

If *resolved_name* is NULL, the function allocates a buffer of the size indicated by _PC_PATH_MAX for storing the real path information in. You're responsible for freeing the memory.

[*sbrk\(\)* \(p. 2702\)](#)

We've corrected the example.

[*sched_param* \(p. 2731\)](#)

The *sched_ss_init_budget* must be smaller than *sched_ss_repl_period*.

***sigaction()* (p. 2948), *SignalAction()* (p. 2977)**

If you set the action for SIGCHLD to SIG_DEF, the signal is ignored, but the process's children can still become zombies. They don't become zombies if you set the action for SIGCHLD to SIG_IGN.

***signal()* (p. 2974)**

If you want to terminate your program from a signal handler, call *_exit()*, not *exit()*.

***strptime()* (p. 3233)**

The classification for this function is POSIX 1003.1 XSI.

***SyncDestroy()* (p. 3289)**

The owner of a locked mutex is able to destroy it.

***SyncTypeCreate()* (p. 3305)**

- You should treat the *sync_t* synchronization object as opaque.
- The names of the members of the *__sync_attr* structure start with two underscores.
- The synchronization object isn't always local to the process.

***tcsetsid()* (p. 3369)**

Both arguments to this function are declared as *const*.

***tell64()* (p. 3373)**

The classification for this function is QNX Neutrino.

***ThreadCreate()* (p. 3402)**

- The names of the members of the *__thread_attr* structure all start with two underscores (*__*).
- If the *__stackaddr* member of the thread attributes structure is NULL, then the *__stacksize* member specifies the *minimum* size of stack to dynamically allocate. If you set *__stacksize* to zero, then a system-defined amount is used.
- If the *__stackaddr* member of the thread attributes structure isn't NULL, then if the *__stacksize* member is less than PTHREAD_STACK_MIN, the call fails with an error of EINVAL.

***TimerTimeout()* (p. 3464), *timer_timeout()* (p. 3442)**

You can OR `TIMER_ABSTIME` into the *flags* argument if you want to set an absolute expiration time.

[*umount\(\)*](#) (p. 3559)

This function sets *errno* if an error occurs.

What's new in QNX Neutrino 6.5.0 Service Pack 1?

New entries

InterruptStatus() (p. 1178)

Determine whether or not interrupts are enabled

procmgr_event_notify_add() (p. 2163), *procmgr_event_notify_delete()* (p. 2168)

Add or remove a request to be notified of system-wide events.

round(), *roundf()*, *roundl()* (p. 2661)

Round a number to the closest integer

Changed content

alloca() (p. 147)

The documentation now describes how the builtin and non-builtin versions of this function behave.

chroot() (p. 295)

This entry now points out that the *path* argument can include a network root (e.g. */net/node_name*).

dladdr() (p. 449)

This function now provides the full path (instead of the base name) of the object, in order to match the behavior on other OSs.

dlopen() (p. 454)

For security reasons, the use of **LD_DEBUG_OUTPUT** with setuid binaries is disabled.

fcntl() (p. 623)

This function indicates an error of **ENOSYS** if the filesystem doesn't support the operation.

flock() (p. 666)

This function indicates an error of **ENOSYS** if the filesystem doesn't support file locks.

[*flockfile\(\)*](#) (p. 669), [*ftrylockfile\(\)*](#) (p. 796), [*funlockfile\(\)*](#) (p. 801)

We've clarified the relationship between these functions.

[*_intr_v86\(\)*](#) (p. 1185)

We've expanded the list of possible errors.

[*lockf\(\)*](#) (p. 1535)

This function indicates an error of ENOSYS if the filesystem doesn't support file locks.

[*mmap\(\)*](#) (p. 1656)

- We've clarified the circumstances under which this function can give an error of ENXIO.
- The calling process must be running as `root` in order to map physical memory.

[*mmap_device_io\(\)*](#) (p. 1666)

On non-x86 systems, the calling process must be running as `root` in order to map device I/O memory.

[*mmap_device_memory\(\)*](#) (p. 1668)

The calling process must be running as `root` in order to map device memory.

[*mount\(\)*](#) (p. 1691)

Note that updating a mount resets the other mount flags to their default values.

[*mount_parse_generic_args\(\)*](#) (p. 1694)

We've added `IMPLIED` to the list of options that this function processes.

[*MsgCurrent\(\)*](#) (p. 1734)

Calling *MsgCurrent()* on the `rcvid` pertaining to a low-priority client no longer causes a priority boost given to your thread by a blocked high-priority thread to be lost.

[*nanospin_calibrate\(\)*](#) (p. 1859)

In order to make startup faster and reduce jitter, *nanospin_calibrate()* now tries to read the calibration data from values stored in the system page. The startup for some boards includes an `-o` option that you can use to specify

the calibration data (100 loop time and overhead) to store in the system page.

[*pci_attach_device\(\)*](#) (p. 1966)

This entry now describes the `PCI_USE_MSI` and `PCI_USE_MSIX` flags and includes an example that shows how you can detect whether or not a device supports MSI or MSI-X.

[*pci_read_config\(\)*](#) (p. 1989), [*pci_read_config8\(\)*](#) (p. 1991), [*pci_read_config16\(\)*](#) (p. 1993), [*pci_read_config32\(\)*](#) (p. 1995)

Note that the *pci_read_config**(*)* functions can return inconsistent data if another process is extensively using the PCI bus.

[*posix_spawn\(\)*](#) (p. 2037), [*posix_spawn_file_actions_addclose\(\)*](#) (p. 2048), [*posix_spawn_file_actions_adddup2\(\)*](#) (p. 2050), [*posix_spawn_file_actions_addopen\(\)*](#) (p. 2052), [*posix_spawn_file_actions_destroy\(\)*](#) (p. 2055), [*posix_spawn_file_actions_init\(\)*](#) (p. 2057), [*posix_spawnattr_addpartid\(\)*](#) (p. 2059), [*posix_spawnattr_addpartition\(\)*](#) (p. 2061), [*posix_spawnattr_destroy\(\)*](#) (p. 2064), [*posix_spawnattr_getcred\(\)*](#) (p. 2066), [*posix_spawnattr_getflags\(\)*](#) (p. 2068), [*posix_spawnattr_getnode\(\)*](#) (p. 2070), [*posix_spawnattr_getpartid\(\)*](#) (p. 2072), [*posix_spawnattr_getpgroup\(\)*](#) (p. 2074), [*posix_spawnattr_getrunmask\(\)*](#) (p. 2076), [*posix_spawnattr_getschedparam\(\)*](#) (p. 2078), [*posix_spawnattr_getschedpolicy\(\)*](#) (p. 2080), [*posix_spawnattr_getsigdefault\(\)*](#) (p. 2082), [*posix_spawnattr_getsigignore\(\)*](#) (p. 2084), [*posix_spawnattr_getsigmask\(\)*](#) (p. 2086), [*posix_spawnattr_getstackmax\(\)*](#) (p. 2088), [*posix_spawnattr_getxflags\(\)*](#) (p. 2090), [*posix_spawnattr_init\(\)*](#) (p. 2093), [*posix_spawnattr_setcred\(\)*](#) (p. 2095), [*posix_spawnattr_setflags\(\)*](#) (p. 2098), [*posix_spawnattr_setnode\(\)*](#) (p. 2101), [*posix_spawnattr_setpgroup\(\)*](#) (p. 2103), [*posix_spawnattr_setrunmask\(\)*](#) (p. 2105), [*posix_spawnattr_setschedparam\(\)*](#) (p. 2107), [*posix_spawnattr_setschedpolicy\(\)*](#) (p. 2109), [*posix_spawnattr_setsigdefault\(\)*](#) (p. 2111), [*posix_spawnattr_setsigignore\(\)*](#) (p. 2113), [*posix_spawnattr_setsigmask\(\)*](#) (p. 2115), [*posix_spawnattr_setstackmax\(\)*](#) (p. 2117), [*posix_spawnattr_setxflags\(\)*](#) (p. 2119), [*posix_spawnnp\(\)*](#) (p. 2037)

We've revised and expanded the documentation for these functions.

[*pthread_setname_np\(\)*](#) (p. 2398)

`_NTO_THREAD_NAME_MAX` is defined in `<sys/neutrino.h>`.

[*readdir\(\)*](#) (p. 2514)

Note that you can use [*ftw\(\)*](#) (p. 798) as a different way to walk a file tree.

[*resmgr_attach\(\)*](#) (p. 2580)

This function indicates an error of EBUSY if an internal resource isn't available.

[*shm_ctl\(\)*](#) (p. 2925), [*shm_ctl_special\(\)*](#) (p. 2931)

We've documented the SHMCTL_ISADMA and SHMCTL_NOX64K flags.

[*spawn\(\)*](#) (p. 3089), [*spawnl\(\)*](#) (p. 3099), [*spawnle\(\)*](#) (p. 3105), [*spawnlp\(\)*](#) (p. 3111), [*spawnlpe\(\)*](#) (p. 3116), [*spawnp\(\)*](#) (p. 3122), [*spawnv\(\)*](#) (p. 3129), [*spawnve\(\)*](#) (p. 3134), [*spawnvp\(\)*](#) (p. 3140), [*spawnvpe\(\)*](#) (p. 3145)

If you use the P_WAIT flag, you need to use a status macro such as *WEXITSTATUS()* to extract information from value returned by the *spawn*()* functions; see “[Status macros](#) (p. 3652)” in the documentation for *wait()*.

[*straddstr\(\)*](#) (p. 3178)

We've added more details about how this function handles null characters.

Errata

[*ChannelCreate\(\)*](#) (p. 278)

Typically your program calls *ChannelCreate()* only once, and all threads block on that channel. We've corrected a call to *MsgReply()* in one of the examples.

[*clock_getcpuclockid\(\)*](#) (p. 307), [*clock_gettime\(\)*](#) (p. 312), [*ClockId\(\)*](#), [*ClockId_r\(\)*](#) (p. 327), [*ClockTime\(\)*](#), [*ClockTime_r\(\)*](#) (p. 333)

We've corrected the description of these functions.

[*dircntl\(\)*](#) (p. 419)

We've corrected the description of what this function returns.

[*dirname\(\)*](#) (p. 424)

If you pass a string of `"/usr/"` to *dirname()*, the string returned is `"/"`, not `"usr"`.

[*exec\(\)*](#) (p. 550), [*execle\(\)*](#) (p. 556), [*execlp\(\)*](#) (p. 562), [*execlpe\(\)*](#) (p. 568), [*execv\(\)*](#) (p. 573), [*execve\(\)*](#) (p. 579), [*execvp\(\)*](#) (p. 585), [*execvpe\(\)*](#) (p. 591)

We've corrected the spelling of EACCES.

[*ftruncate\(\)*](#), [*ftruncate64\(\)*](#) (p. 793)

These functions can indicate an error of `ENOMEM` if there wasn't enough memory to change the size of a shared memory object.

[`ftw\(\)`](#) (p. 798)

We've corrected the spelling of `EACCES`.

[`getrlimit\(\)`](#) (p. 942)

`RLIMIT_FSIZE` and `RLIMIT_MEMLOCK` aren't supported.

[`getsockopt\(\)`](#) (p. 962)

We've corrected the call to [`memset\(\)`](#) (p. 1620) in the example.

[`iofunc\_check\_access\(\)`](#) (p. 1230)

We've corrected the spelling of `EACCES`.

[`iofunc\_stat\_default\(\)`](#) (p. 1355)

This function returns -1, to indicate to the resource manager library that it should return one part from the `ctp->iov` structure (see [`resmgr\_context\_t`](#) (p. 2598)) to the client.

[`mem\_offset\(\)`](#), [`mem\_offset64\(\)`](#) (p. 1600)

We've corrected the description of these functions. You must specify `NOFD` for the `fd` argument; to get the offset and length of a mapped typed memory block, use [`posix\_mem\_offset\(\)`](#) or [`posix\_mem\_offset64\(\)`](#) (p. 2030).

[`mount\_parse\_generic\_args\(\)`](#) (p. 1694)

We've corrected the names of the `_MOUNT_NOATIME`, `_MOUNT_NOCREAT`, and `_MOUNT_NOSUID` flags.

[`mq\_open\(\)`](#) (p. 1709)

You need to include `<fcntl.h>` if you want to use `O_CREAT`, `O_EXCL`, or `O_NONBLOCK` when you call this function.

[`munmap\(\)`](#) (p. 1832)

The ability to [`munmap\(\)`](#) just a part of an area mapped with [`mmap\(\)`](#) (p. 1656) was introduced in QNX Neutrino 6.3.2.

[`nftw\(\)`](#) (p. 1888), [`nftw64\(\)`](#) (p. 1888)

- We've corrected the spelling of `EACCES`.
- We've corrected the name of the `<ftw.h>` header file.

[posix_spawn\(\)](#) (p. 2037), [posix_spawn_file_actions_addclose\(\)](#) (p. 2048),
[posix_spawn_file_actions_adddup2\(\)](#) (p. 2050), [posix_spawn_file_actions_addopen\(\)](#) (p. 2052),
[posix_spawn_file_actions_destroy\(\)](#) (p. 2055), [posix_spawn_file_actions_init\(\)](#) (p. 2057),
[posix_spawnattr_addpartid\(\)](#) (p. 2059), [posix_spawnattr_addpartition\(\)](#) (p. 2061),
[posix_spawnattr_destroy\(\)](#) (p. 2064), [posix_spawnattr_getcred\(\)](#) (p. 2066), [posix_spawnattr_getflags\(\)](#)
(p. 2068), [posix_spawnattr_getnode\(\)](#) (p. 2070), [posix_spawnattr_getpartid\(\)](#) (p. 2072),
[posix_spawnattr_getpgroup\(\)](#) (p. 2074), [posix_spawnattr_getrunmask\(\)](#) (p. 2076),
[posix_spawnattr_getschedparam\(\)](#) (p. 2078), [posix_spawnattr_getschedpolicy\(\)](#) (p. 2080),
[posix_spawnattr_getsigdefault\(\)](#) (p. 2082), [posix_spawnattr_getsigignore\(\)](#) (p. 2084),
[posix_spawnattr_getsigmask\(\)](#) (p. 2086), [posix_spawnattr_getstackmax\(\)](#) (p. 2088),
[posix_spawnattr_getxflags\(\)](#) (p. 2090), [posix_spawnattr_init\(\)](#) (p. 2093), [posix_spawnattr_setcred\(\)](#)
(p. 2095), [posix_spawnattr_setflags\(\)](#) (p. 2098), [posix_spawnattr_setnode\(\)](#) (p. 2101),
[posix_spawnattr_setpgroup\(\)](#) (p. 2103), [posix_spawnattr_setrunmask\(\)](#) (p. 2105),
[posix_spawnattr_setschedparam\(\)](#) (p. 2107), [posix_spawnattr_setschedpolicy\(\)](#) (p. 2109),
[posix_spawnattr_setsigdefault\(\)](#) (p. 2111), [posix_spawnattr_setsigignore\(\)](#) (p. 2113),
[posix_spawnattr_setsigmask\(\)](#) (p. 2115), [posix_spawnattr_setstackmax\(\)](#) (p. 2117),
[posix_spawnattr_setxflags\(\)](#) (p. 2119), [posix_spawnnp\(\)](#) (p. 2037)

- We've corrected the synopsis for *posix_spawn()* and *posix_spawnnp()*, and corrected the list of reasons why these functions might indicate an error of EINVAL.
- These functions return error codes instead of setting *errno*.
- These functions are declared in `<spawn.h>`.
- It isn't safe to call *posix_spawn()* or *posix_spawnnp()* from an interrupt handler.

[posix_typed_mem_open\(\)](#) (p. 2124)

You need to include `<fcntl.h>` in order to get the definitions of `O_RDONLY`, etc.

[pthread_getcpuclockid\(\)](#) (p. 2295)

We've corrected the description of this function.

[rename\(\)](#) (p. 2560)

We've corrected the spelling of EACCES.

[SchedGet\(\)](#) (p. 2774)

This routine returns an actual scheduling policy value, not SCHED_ADJTOHEAD or SCHED_ADJTOTAIL.

[*sem_open\(\)*](#) (p. 2814)

You need to include `<fcntl.h>` if you want to use `O_CREAT` or `O_EXCL` when you call this function.

[*sem_unlink\(\)*](#) (p. 2825)

We've corrected the spelling of EACCES.

[*setdomainname\(\)*](#) (p. 2849)

It isn't safe to call this function from an interrupt handler.

[*setrlimit\(\)*](#) (p. 2899)

RLIMIT_FSIZE and RLIMIT_MEMLOCK aren't supported.

[*shm_ctl_special\(\)*](#) (p. 2931)

Calling *shm_ctl_special()* with a *special* argument of 0 **isn't** equivalent to calling *shm_ctl()*.

[*spawn\(\)*](#) (p. 3089), [*spawnl\(\)*](#) (p. 3099), [*spawnle\(\)*](#) (p. 3105), [*spawnlp\(\)*](#) (p. 3111), [*spawnlpe\(\)*](#) (p. 3116), [*spawnp\(\)*](#) (p. 3122), [*spawnv\(\)*](#) (p. 3129), [*spawnve\(\)*](#) (p. 3134), [*spawnvp\(\)*](#) (p. 3140), [*spawnvpe\(\)*](#) (p. 3145)

We've corrected the spelling of EACCES.

[*stat\(\)*](#) (p. 3164)

S_IFMT is a mask for the file-type bits in *st_mode*.

[*SyncTypeCreate\(\)*](#) (p. 3305)

We've corrected the name of the `__sync_attr` structure.

[*tmpfile\(\)*](#) (p. 3478)

We've corrected the spelling of EACCES.

[*trace_logf\(\)*](#) (p. 3502), [*trace_nlogf\(\)*](#) (p. 3506), [*trace_vnlogf\(\)*](#) (p. 3508)

It isn't safe to call these functions from an interrupt handler if the formatting codes or parameters include floating-point elements.

[*wscpy\(\)*](#) (p. 3680), [*wscncpy\(\)*](#) (p. 3691), [*wcstoul\(\)*](#) (p. 3713), [*wcstoull\(\)*](#) (p. 3713), [*wcsxfrm\(\)*](#) (p. 3715), [*wprintf\(\)*](#) (p. 3736), [*wscanf\(\)*](#) (p. 3748)

We've corrected the synopsis.

What's new in QNX Neutrino 6.5.0?

New entries

acoshl() (p. 120)

Compute the inverse hyperbolic cosine

acosl() (p. 118)

Compute the arccosine of an angle

asinh() (p. 159)

Compute the inverse hyperbolic sine

asinl() (p. 157)

Compute the arcsine of an angle

atanhl() (p. 167)

Compute an inverse hyperbolic tangent

atanl() (p. 163)

Compute the arctangent of an angle

coshl() (p. 379)

Compute the hyperbolic cosine

cosl() (p. 377)

Compute the cosine of an angle

devctlv() (p. 403)

Control a device, using I/O vectors to pass data to and from the driver

erfcl() (p. 534)

Complementary error function

expl() (p. 601)

Compute the exponential function of a number

fmodl() (p. 675)

Compute a residue, using floating-point modular arithmetic

[*fseeko64\(\)*](#) (p. 772)

Change the current position of a stream

[*ftello64\(\)*](#) (p. 789)

Return the current position of a stream

[*getpagesizes\(\)*](#) (p. 909), [*getpagesizes64\(\)*](#) (p. 909)

Get the available page sizes

[*getpeereid\(\)*](#) (p. 913)

Get the effective credentials of a UNIX-domain peer

[*ioctl_socket\(\)*](#) (p. 1198)

An optimized version of *ioctl()* that provides special handling for some commands that use embedded pointers

[*ldexpl\(\)*](#) (p. 1506)

Multiply a floating-point number by an integral power of 2

[*llabs\(\)*](#) (p. 1525)

Calculate the absolute value of a long long integer

[*log1pl\(\)*](#) (p. 1543)

Log(1+x)

[*log10l\(\)*](#) (p. 1541)

Compute the logarithm (base 10) of a number

[*logl\(\)*](#) (p. 1539)

Compute the natural logarithm of a number

[*ncurses*](#) (p. 1873)

Control a terminal's display

[*_NTO_TRACE_GETEVENT_C\(\)*](#), [*_NTO_TRACE_GETEVENT\(\)*](#), [*_NTO_TRACE_GETCPU\(\)*](#),
[*_NTO_TRACE_SETEVENT_C\(\)*](#), [*_NTO_TRACE_SETEVENT\(\)*](#)

Macros for use with the header of a trace event. For more information, see
“[Macros for use with the event header](#) (p. 3517)” in the entry for *TraceEvent()*.

[*powl\(\)*](#) (p. 2129)

Raise a number to a given power

***pthread_mutex_timedlock_monotonic()* (p. 2324)**

This function is similar to *pthread_mutex_timedlock()*, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

***pthread_mutex_wakeup_np()* (p. 2331)**

Unblock any threads that are waiting on a mutex

***pthread_mutexattr_getwakeup_np()* (p. 2345)**

Get the wakeup attribute of a mutex

***pthread_mutexattr_setwakeup_np()* (p. 2359)**

Set the wakeup attribute of a mutex

***pthread_setschedprio()* (p. 2402)**

Set a thread's priority

***pthread_timedjoin_monotonic()* (p. 2433)**

This function is similar to *pthread_timedjoin()*, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

***remainder()* (p. 2554)**

Compute the floating point remainder

***rintl()* (p. 2654)**

Round to the nearest integral value

***sem_timedwait_monotonic()* (p. 2820)**

This function is similar to *sem_timedwait()*, but it uses `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

***shutdown_system()* (p. 2944)**

Shut down the system; note that you can customize this function by providing your own callouts:

- *shutdown_classify()* (p. 2945)
- *shutdown_display()* (p. 2945)
- *shutdown_done()* (p. 2946)

- [*shutdown_error\(\)*](#) (p. 2946)
- [*shutdown_process\(\)*](#) (p. 2946)
- [*shutdown_progress\(\)*](#) (p. 2946)
- [*shutdown_prompt\(\)*](#) (p. 2946)

[*sinh\(\)*](#) (p. 3026)

Compute the hyperbolic sine

[*sin\(\)*](#) (p. 3024)

Calculate the sine of an angle

[*sqrt\(\)*](#) (p. 3153)

Calculate the nonnegative square root of a number

[*strerror_r\(\)*](#) (p. 3200)

Convert an error number into an error message (reentrant)

[*strtof\(\)*](#) (p. 3253), [*strtold\(\)*](#) (p. 3253)

Convert a string into a float and long double, respectively.

[*tanh\(\)*](#) (p. 3336)

Calculate the hyperbolic tangent

[*tan\(\)*](#) (p. 3334)

Calculate the tangent of an angle

[*trace_func_enter\(\)*](#) (p. 3492)

Insert a trace event for the entry to a function

[*trace_func_exit\(\)*](#) (p. 3494)

Insert a trace event for the exit from a function

[*trace_here\(\)*](#) (p. 3496)

Insert a trace event for the current address

[*trace_logb\(\)*](#) (p. 3498)

Insert a user combine trace event

[*trace_logbc\(\)*](#) (p. 3500)

Insert a trace event of an arbitrary class and type with arbitrary data

[trace_logf\(\)](#) (p. 3502)

Insert a user string trace event

[trace_logi\(\)](#) (p. 3504)

Insert a user simple trace event

[trace_nlogf\(\)](#) (p. 3506)

Insert a user string trace event, specifying a maximum string length

[trace_vnlogf\(\)](#) (p. 3508)

Insert a user string trace event, using a variable argument list

Changed content

[acos\(\)](#) (p. 118), [acosf\(\)](#) (p. 118), [acosl\(\)](#) (p. 118)

For finite values not in the range $[-1, 1]$, these functions return NaN. The return value for $\pm\infty$ is NaN.

[acosh\(\)](#) (p. 120), [acoshf\(\)](#) (p. 120), [acoshl\(\)](#) (p. 120)

For finite values of $x < 1$, these functions return NaN. The return value when x is $-\infty$ is NaN.

[asin\(\)](#) (p. 157), [asinf\(\)](#) (p. 157), [asinhl\(\)](#) (p. 157)

For finite values not in the range $[-1, 1]$, these functions return NaN. The return value for $\pm\infty$ is NaN.

[atan\(\)](#) (p. 163), [atanf\(\)](#) (p. 163), [atanhl\(\)](#) (p. 163)

For finite values of $|x| > 1$, these functions return NaN. The return value for $\pm\infty$ is NaN.

[aio_cancel\(\)](#) (p. 124)

The documentation now says that if *aiocbptr* isn't NULL, *aio_cancel()* ignores the *fildev* argument and attempts to cancel the I/O operation specified by the *aio* control block. The operation isn't canceled if it's already in progress.

This function indicates an error of EINVAL if the control block that *aiocbptr* points to isn't valid (i.e. it hasn't yet been used in any call to *aio_read()* or *aio_write()*).

[aio_read\(\)](#) (p. 131), [aio_write\(\)](#) (p. 138)

The documentation now says that the asynchronous operation is submitted at the scheduling priority of the thread minus *aiocbp-> aio_reqprio*.

If synchronized I/O is enabled on the file associated with *aiocbptr->aio_fildes*, these functions behave in accordance with the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

[*calloc\(\)* \(p. 255\)](#), [*iofunc_lock_calloc\(\)* \(p. 1279\)](#), [*malloc\(\)* \(p. 1569\)](#), [*memalign\(\)* \(p. 1603\)](#), [*realloc\(\)* \(p. 2527\)](#), [*valloc\(\)* \(p. 3610\)](#)

Because the *malloc()* implementation uses signed, 32-bit integers to represent the size internally, you can't allocate more than 2 GB in a single allocation. If the size is greater than 2 GB, these functions indicate an error of ENOMEM.

[*bind\(\)* \(p. 208\)](#)

Before calling *bind()* on an AF_INET socket, set the *af_family* member of the sockaddr structure to AF_INET. Up until QNX Neutrino 6.4.0, a value of 0 was accepted and assumed to be this value.

[*CACHE_FLUSH\(\)* \(p. 246\)](#), [*CACHE_INVALID\(\)* \(p. 253\)](#)

Before using the *CACHE_**() macros on ARM platforms, the calling thread must obtain I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

[*ChannelCreate\(\)* \(p. 278\)](#)

We've added some sample code that you can use to check for stale _PULSE_CODE_COIDDEATH pulses.

[*ClockTime\(\)* \(p. 333\)](#)

If you call *ClockTime()* to set the time of day, the kernel checks to see if the SYSPAGE_ENTRY(qtime) ->boot_time field is zero. If it is, the kernel sets it to the appropriate value.

[*confstr\(\)* \(p. 344\)](#)

We've described the _CS_LOCALE configuration string.

[*ConnectAttach\(\)* \(p. 351\)](#)

We've added EINVAL, EMFILE, and ENOREMOTE to the list of error codes.

[*ConnectClientInfo\(\)* \(p. 356\)](#)

The *sid* member of the _client_info structure is used internally by Qnet.

[ConnectServerInfo\(\)](#) (p. 372)

We've documented the *flags* field of the `_server_info` structure.

If the *info* argument is `NULL`, `ConnectServerInfo()` ignores connections with dead servers and skips to the next coid. If *info* is non-`NULL`, the function fills in the `_server_info` structure; for connections with dead servers, it turns on the `_NTO_COF_DEAD` bit in the *flags* field of the structure.

[cos\(\)](#) (p. 377), [cosf\(\)](#) (p. 377), [cosl\(\)](#) (p. 377)

When *x* is $\pm\text{Inf}$, these functions return NaN.

[daemon\(\)](#) (p. 394)

The High Availability Manager now automatically switches to monitoring the new process that `daemon()` creates, if the original process was a self-attached entity.

[dlopen\(\)](#) (p. 454)

RTLD_LAZY is now implemented. For details about it, see “Lazy loading” in the Compiling and Debugging chapter of the QNX Neutrino *Programmer's Guide*.

[erfc\(\)](#) (p. 534), [erfcf\(\)](#) (p. 534), [erfcl\(\)](#) (p. 534)

For a correct value that would cause an underflow, these functions return 0.0.

[exec\(\)](#) (p. 550), [execle\(\)](#) (p. 556), [execlp\(\)](#) (p. 562), [execlpe\(\)](#) (p. 568), [execv\(\)](#) (p. 573), [execve\(\)](#) (p. 579), [execvp\(\)](#) (p. 585), [execvpe\(\)](#) (p. 591)

- These functions can give an error of ETXTBSY if the text file that you're trying to execute is busy (e.g. it might be open for writing).
- The documentation for these functions now mentions that any memory locked by the calling process is unlocked.

[exp\(\)](#) (p. 601), [expf\(\)](#) (p. 601), [expl\(\)](#) (p. 601)

For a correct value that would cause an underflow, these functions return 0.0.

[fclose\(\)](#) (p. 618)

We've described the main error codes associated with this function.

[fcntl\(\)](#) (p. 623)

This function indicates an error of EINVAL if the *cmd* argument is invalid.

The documentation now includes the following values for *cmd*:

- F_ALLOCS64
- F_FREES64
- F_GETLK64
- F_SETLK64
- F_SETLKW64

[*fflush\(\)*](#) (p. 641)

We've described the main error codes associated with this function.

[*fgetc\(\)*](#) (p. 645)

We've described the main error codes associated with this function.

[*fork\(\)*](#) (p. 685)

The documentation for this function now mentions that the child process doesn't inherit any memory locks from the parent.

[*fpathconf\(\)*](#) (p. 699), [*pathconf\(\)*](#) (p. 1936)

We've described the following:

- _PC_ASYNC_IO
- _PC_LINK_DIR
- _PC_PRIO_IO
- _PC_SYNC_IO

[*fputc\(\)*](#) (p. 704)

We've described the main error codes associated with this function.

[*freopen\(\)*](#) (p. 721)

This function indicates an error of EBADF if the underlying file descriptor is invalid or doesn't support the requested mode change.

We've listed the changes of mode that the Neutrino version of this function permits.

[*fseek\(\)*](#), [*fseeko\(\)*](#) (p. 772)

We've listed the errors that these functions can indicate.

As a QNX Neutrino extensions, these functions give an error of ENOSYS if the underlying device is incapable of seeking.

[fstatvfs\(\)](#) (p. 783), [statvfs\(\)](#) (p. 3170)

- The values of the *f_files*, *f_ffree*, and *f_favail* members of the `statvfs` structure depend on the filesystem.
- We've updated the list of possible bits in the *f_flags* member.
- The filesystem ID is currently the device ID.

[getsockopt\(\)](#) (p. 962)

This entry now includes a section on setting the [keepalive timing](#) (p. 968).

[glob\(\)](#) (p. 999)

We've documented the following flags (which are NetBSD extensions):

- `GLOB_PERIOD`
- `GLOB_MAGCHAR`
- `GLOB_ALTDIRFUNC`
- `GLOB_BRACE`
- `GLOB_NOMAGIC`
- `GLOB_TILDE`
- `GLOB_NO_DOTDIRS`
- `GLOB_LIMIT`

[inbe16\(\)](#) (p. 1063), [inle16\(\)](#) (p. 1063), [inbe32\(\)](#) (p. 1067), [inle32\(\)](#) (p. 1067)

Note that these macros access the specified port more than once if endian conversion is necessary. This could be a problem on some hardware.

[InterruptAttach\(\)](#) (p. 1147), [InterruptAttachEvent\(\)](#) (p. 1154)

Note that it isn't safe to use floating-point operations in an Interrupt Service Routine (ISR).

[InterruptMask\(\)](#) (p. 1176), [InterruptUnmask\(\)](#) (p. 1181)

These functions now give an error of `ESRCH` if the *id* parameter isn't something returned by [InterruptAttach\(\)](#) (p. 1147) or [InterruptAttachEvent\(\)](#) (p. 1154), or -1.

[InterruptDetach\(\)](#) (p. 1162)

The thread that detaches an interrupt handler must be in the same process as the thread that attached it, or else this function indicates an error of `EPERM`.

[ioctl\(\)](#) (p. 1198)

We've made *ioctl()* compatible with other UNIX-based systems by enabling support for embedded pointers. This means that you no longer have to change calls to *ioctl()* into calls to *ioctl_socket()* in networking applications. As a result of this change, *ioctl()* can now indicate an error of **ENOBUFFS** if there isn't enough memory available to copy the data that the embedded pointers refer to.

[*ionotify\(\)*](#) (p. 1386)

The documentation now states that this function enforces a restriction of one notification per message queue or resource manager. If a notification was already armed for the specified resource, *ionotify()* indicates an error of **EBUSY**.

[*ldexp\(\)*](#) (p. 1506), [*ldexpf\(\)*](#) (p. 1506), [*ldexpl\(\)*](#) (p. 1506)

For a correct value that would cause an underflow, these functions return 0.0.

[*log\(\)*](#) (p. 1539), [*logf\(\)*](#) (p. 1539), [*logl\(\)*](#) (p. 1539)

For finite values of *x* less than zero, these functions return NaN. The return value when *x* is -Inf is NaN.

[*log10\(\)*](#) (p. 1541), [*log10f\(\)*](#) (p. 1541), [*log10l\(\)*](#) (p. 1541)

For finite values of *x* less than zero, these functions return NaN. The return value when *x* is -Inf is NaN.

[*log1p\(\)*](#) (p. 1543), [*log1pf\(\)*](#) (p. 1543), [*log1pl\(\)*](#) (p. 1543)

For finite values of *x* less than -1, these functions return NaN. The return value when *x* is -Inf is NaN.

[*mallopt\(\)*](#) (p. 1575)

We've renamed **MALLOC_FILLAREA** to **MALLOC_CKBOUNDS**, and we've added more information about the **MALLOC_CKALLOC**, **MALLOC_DUMP_LEAKS**, **MALLOC_EVENTBTDEPTH**, **MALLOC_FREE_CHECK**, **MALLOC_START_TRACING**, **MALLOC_STATS**, **MALLOC_TRACEBTDEPTH**, **MALLOC_TRACEMAX**, **MALLOC_TRACEMIN**, **MALLOC_TRACING**, and **MALLOC_VERIFY_ON** options.

[*mem_offset\(\)*](#) (p. 1600)

If the offset is greater than **INT_MAX**, *mem_offset()* gives an error of **EOVERFLOW**. Use *mem_offset64()* if this happens.

[*message_attach\(\)*](#) (p. 1624)

Note that although the *low* and *high* arguments are of type `int`, *message_attach()* stores the values internally as `short`.

[`mkdir\(\)`](#) (p. 1636)

We've described the behavior for the `S_ISGID` and `S_ISVTX` bits.

Note that not all filesystems support the creation of directories.

[`mlock\(\)`](#) (p. 1651), [`mlockall\(\)`](#) (p. 1653), [`munlock\(\)`](#) (p. 1828), [`munlockall\(\)`](#) (p. 1830)

For more information about locking, superlocking, and unlocking memory, see “Locking memory” in the Process Manager chapter of the *System Architecture* guide.

There's no limit on the amount of memory that a process may lock, other than the amount of physical memory in the system.

[`mmap\(\)`](#) (p. 1656)

The *off* argument to *mmap()* can be a physical address (e.g. for mapping a device's registers in a resource manager).

[`mq\_open\(\)`](#) (p. 1709), [`mq\_unlink\(\)`](#) (p. 1728)

These functions indicate an error of `ENOSYS` if the message queue manager (`mq` or `mqueue`) isn't running.

[`MsgCurrent\(\)`](#) (p. 1734)

The documentation now points out that if you call *MsgCurrent()* on the *rcvid* pertaining to a low-priority client, then the priority boost given to your thread by a blocked high-priority thread can be lost.

[`MsgSend\(\)`](#) (p. 1781), [`MsgSend\_r\(\)`](#) (p. 1781), [`MsgSendnc\(\)`](#) (p. 1785), [`MsgSendnc\_r\(\)`](#) (p. 1785), [`MsgSendPulse\(\)`](#) (p. 1789), [`MsgSendPulse\_r\(\)`](#) (p. 1789), [`MsgSendsv\(\)`](#) (p. 1793), [`MsgSendsv\_r\(\)`](#) (p. 1793), [`MsgSendsvnc\(\)`](#) (p. 1797), [`MsgSendsvnc\_r\(\)`](#) (p. 1797), [`MsgSendvs\(\)`](#) (p. 1801), [`MsgSendvs\_r\(\)`](#) (p. 1801), [`MsgSendvsnc\(\)`](#) (p. 1805), [`MsgSendvsnc\_r\(\)`](#) (p. 1805), [`MsgSendvsv\(\)`](#) (p. 1809), [`MsgSendvsv\_r\(\)`](#) (p. 1809), [`MsgSendvsvnc\(\)`](#) (p. 1813), [`MsgSendvsvnc\_r\(\)`](#) (p. 1813)

These functions can indicate an error of `EHOSTDOWN` or `EHOSTUNREACH` if (for example) a send across Qnet failed.

[`MsgSendPulse\(\)`](#) (p. 1789)

- If you specify a priority of -1, the function uses the calling thread's priority as the pulse's priority.

- *MsgSendPulse()* gives an error of EINVAL if the specified priority is invalid (e.g. 0).

[*pow\(\)* \(p. 2129\)](#), [*powf\(\)* \(p. 2129\)](#), [*powl\(\)* \(p. 2129\)](#)

For finite values of *x* less than 0 and finite non-integer values of *y*, these functions return NaN. The return value for a correct value that would cause underflow is 0.0.

[*procmgr_daemon\(\)* \(p. 2156\)](#)

Note that the High Availability Manager (HAM) automatically monitors processes detached using *procmgr_daemon()* because they're in session 1.

[*pthread_cond_timedwait\(\)* \(p. 2265\)](#)

We've changed the example so that it uses CLOCK_MONOTONIC, which isn't affected by changes to the system time (unlike the default, CLOCK_REALTIME).

[*pthread_getschedparam\(\)* \(p. 2299\)](#)

As a QNX extension, the *policy* argument to *pthread_getschedparam()* can be NULL, in which case the function doesn't try to store the scheduling policy.

[*pthread_mutex_lock\(\)* \(p. 2318\)](#)

As a QNX Neutrino extension, this function can now return EINTR if the thread was unblocked by a call to [*pthread_mutex_wakeup_np\(\)* \(p. 2331\)](#).

[*pthread_mutexattr_setrecursive\(\)* \(p. 2355\)](#)

The [*pthread_mutexattr_settype\(\)* \(p. 2357\)](#) function provides a more portable way to make a mutex recursive.

[*pthread_rwlock_rdlock\(\)* \(p. 2369\)](#)

If another thread is currently waiting to acquire the read-write lock for writing, *pthread_rwlock_rdlock()* blocks. This can cause a deadlock if a thread attempts to recursively acquire the lock for reading, while another thread is trying to acquire the lock for writing.

[*pthread_setname_np\(\)* \(p. 2398\)](#)

This function gives an error of EBUSY when you're trying to name a thread other than the calling thread, and another program already has write access to the process's entry in the /proc filesystem.

***read()* (p. 2502)**

This function indicates an error of EIO if the filesystem resides on a removable media device, and the media has been forcibly removed.

***readdir()* (p. 2514)**

We've added more information about the `dirent_extra` and `dirent_extra_stat` structures that filesystems can provide as extra data.

***regexec()* (p. 2549)**

We've corrected the naming of the members from `rm_sp` and `rm_ep` to be `rm_so` and `rm_eo` respectively.

***remainder()* (p. 2554), *remainderf()* (p. 2554), *remainderl()* (p. 2554)**

If *x* is infinite or *y* is zero and the other is non-NaN, these functions return NaN.

***scalbn()* (p. 2707), *scalbnf()* (p. 2707)**

When a correct value would cause an underflow, these functions return 0.0.

***scanf()* (p. 2713)**

A - in the scanlist for the [conversion specifier that isn't the first character, nor the second where the first character is a ^, nor the last character, defines a range of characters to be matched. This range consists of characters numerically greater than or equal to the character before the -, and numerically less than or equal to the character after the -.

For the p conversion specifier, if the sequence contains more than 8 digits, the conversion is performed only on the last 8 digits in the sequence.

***send()* (p. 2829), *sendmsg()* (p. 2833), *sendto()* (p. 2837)**

These functions support the MSG_NOSIGNAL flag. Set it if you don't want to raise a SIGPIPE signal when the other end breaks the connection.

***shm_ctl()* (p. 2925), *shm_ctl_special()* (p. 2931)**

If you specify SHMCTL_PHYS in the flags, then *paddr* and *size* must be even multiples of the page size (`sysconf(_SC_PAGE_SIZE)`).

***sigaction()* (p. 2948), *signal()* (p. 2974), *SignalAction()* (p. 2977)**

Note that it isn't safe to use floating-point operations in a signal handler.

***SignalAction()* (p. 2977)**

QNX Neutrino uses the top eight signals for special purposes. They're always masked, and attempts to unmask them are ignored. They include named signals (e.g., SIGSELECT) and unnamed signals.

[*sigtimedwait\(\)*](#) (p. 3016)

This entry now mentions that the CLOCK_MONOTONIC clock is used to measure the time interval.

[*sin\(\)*](#) (p. 3024), [*sinf\(\)*](#) (p. 3024), [*sinl\(\)*](#) (p. 3024)

If *x* is +/-Inf, these functions return NaN.

[*sopen\(\)*](#) (p. 3083)

We've added EEXIST to the list of errors that this function can indicate.

[*spawn\(\)*](#) (p. 3089), [*spawnp\(\)*](#) (p. 3122)

- We've added more details about working with the *fd_count* and *fd_map* arguments to set up the file descriptors in the child process; see “[Mapping file descriptors](#) (p. 3092)” in the entry for *spawn()*.
- We've expanded the list of possible errors:
 - ETXTBSY — the text file that you're trying to execute is busy (e.g. it might be open for writing).
 - EINVAL — an argument is invalid (e.g. *argv[0]* is NULL).
 - EINTR — the function was interrupted by a signal.

[*spawnl\(\)*](#) (p. 3099), [*spawnle\(\)*](#) (p. 3105), [*spawnlp\(\)*](#) (p. 3111), [*spawnlpe\(\)*](#) (p. 3116), [*spawnv\(\)*](#) (p. 3129), [*spawnve\(\)*](#) (p. 3134), [*spawnvp\(\)*](#) (p. 3140), [*spawnvpe\(\)*](#) (p. 3145)

We've expanded the list of possible errors:

- ETXTBSY — the text file that you're trying to execute is busy (e.g. it might be open for writing).
- EINVAL — an argument is invalid. For example, *argv[0]* or *arg0* (depending on which *spawn*()* function you use) is NULL, or the value of the *mode* argument is invalid.
- EINTR — the function was interrupted by a signal.
- ECHILD — the value of the *mode* argument is P_WAIT, and the child process terminated before the *waitpid()* was completed.

[*sqrt\(\)*](#) (p. 3153), [*sqrthf\(\)*](#) (p. 3153), [*sqrthl\(\)*](#) (p. 3153)

For finite values of $x < -0$, these functions return NaN. The return value when x is $-\text{Inf}$ is NaN.

SyncCtl() (p. 3286)

There's a new *cmd* argument, `_NTO_SCTL_MUTEX_WAKEUP`, that's used to wake up any threads that are blocked on the specified mutex; see [*pthread_mutex_wakeup_np\(\)*](#) (p. 2331).



Don't use the `_NTO_SCTL_MUTEX_WAKEUP` command directly.

SyncMutexEvent() (p. 3291)

If you call *SyncMutexEvent()* with a NULL event, the function deletes any existing event registration.

sysconf() (p. 3308)

We've described the following variables:

- `_SC_AIO_PRIO_DELTA_MAX`
- `_SC_DELAYTIMER_MAX`
- `_SC_GETGR_R_SIZE_MAX`
- `_SC_GETPW_R_SIZE_MAX`
- `_SC_SEM_NSEMS_MAX`
- `_SC_SIGQUEUE_MAX`
- `_SC_THREAD_STACK_MIN`
- `_SC_TZNAME_MAX`

SYSPAGE_ENTRY() (p. 3326)

If you call *ClockTime()* to set the time of day, the kernel checks to see if the `SYSPAGE_ENTRY(qtime) -> boot_time` field is zero. If it is, the kernel sets it to the appropriate value.

tan() (p. 3334), ***tanf()*** (p. 3334), ***tanh()*** (p. 3336)

For a correct value that would cause an underflow, these functions return 0.0.

ThreadCtl() (p. 3408)

- Not only does a `_NTO_TCTL_IO` command request I/O privileges, it also superlocks the process's memory.
- A lack of I/O privileges may cause a SIGILL signal.

[thread_pool_attr_t](#) (p. 3388)

This structure — described in the entry for [thread_pool_create\(\)](#) (p. 3387) — now has a *tid_name* entry that you can use to specify a name to use for the threads in the pool.

[timer_create\(\)](#) (p. 3427)

This function fails if the clock ID corresponds to the CPU-time clock of a process or thread different from the process or thread invoking the function.

[timer_settime\(\)](#) (p. 3439)

The documentation now says that this function rounds up time values that are between two consecutive nonnegative integer multiples of the resolution of the specified timer to the larger multiple of the resolution.

[tmpnam\(\)](#) (p. 3480)

After TMP_MAX names have been generated, this function reuses names that have been previously generated:

- If a file with the new name doesn't exist, the new name is returned.
- If a file with the new name does exist, a default pathname with a basename of 000000 is returned.

[TraceEvent\(\)](#) (p. 3510)

We've updated and expanded the entry for this function:

- We've updated the list of event classes.
- In order to set up the dynamic rules filter, your application must call [ThreadCtl\(\)](#) (p. 3408) with the _NTO_TCTL_IO flag to get I/O privileges.
- We've expanded the description of the *event_data_t* structure. Note that you need to provide space for the *data_array* member of this structure.
- We've described the *_NTO_TRACE_GET*()* and *_NTO_TRACE_SET*()* macros, which you can use to work with event headers.

[traceparser\(\)](#) (p. 3524), [traceparser_cs\(\)](#) (p. 3526), [traceparser_cs_range\(\)](#) (p. 3529), [traceparser_debug\(\)](#) (p. 3531), [traceparser_destroy\(\)](#) (p. 3533), [traceparser_get_info\(\)](#) (p. 3534), [traceparser_init\(\)](#) (p. 3537)

We've expanded the entries for these functions, which you'll use if you're creating your own utility for parsing trace data (as an alternative to *traceprinter*).

***tzset()* (p. 3547)**

This function assumes the time zone is UTC0 if neither the **TZ** environment variable nor the `_CS_TIMEZONE` configuration string is set.

Errata***clock_settime()* (p. 319)**

An error of `EINVAL` could mean that the `tv_sec` member of the `timespec` structure is set to -1.

***getopt()* (p. 903)**

We've corrected the "Returns" section of this entry.

***InterruptAttach()* (p. 1147)**

This function currently ignores the *size* argument.

***malloc()* (p. 1569)**

- The default value for **MALLOC_ARENA_CACHE_MAXBLK** is 12, not 8.
- We've corrected the description of **MALLOC_MMAP_NOZERO**. If this environment variable is nonzero, then when *malloc()* needs to expand the heap, it specifies the `MAP_NOINIT` flag when it calls *mmap()* (p. 1656); if the physical memory being mapped was previously unmapped with `UNMAP_INIT_OPTIONAL`, then the POSIX requirement that the memory be zeroed is relaxed. This environment variable has been supported on all architectures since QNX Neutrino 6.3.2.

***mprotect()* (p. 1699)**

In QNX Neutrino 6.4.0 and later, the *addr* argument isn't const.

***mq_open()* (p. 1709)**

We've corrected the description of how this function interprets the *name* argument.

***MsgDeliverEvent()* (p. 1736)**

We've corrected the `server.c` part of the example so that it handles the `IO_CONNECT` message sent because of the usage of *name_attach()*.

***MsgReceive()* (p. 1760), *MsgReply()* (p. 1775), *MsgSend()* (p. 1781), *MsgSendnc()* (p. 1785)**

The value of `SSIZE_MAX` is $2^{31} - 1$, not $2^{32} - 1$.

[*MsgSendPulse\(\)*](#) (p. 1789)

If you send a pulse across the network, the calling thread is blocked (STATE_NET_SEND) while the pulse is delivered to the local Qnet manager, which transmits it to the remote Qnet. The remote Qnet does a *MsgSendPulse()* on behalf of the caller, and the return value is transmitted back, at which point the calling thread is unblocked.

[*munlockall\(\)*](#) (p. 1830)

The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2, but the documentation wasn't updated.

[*posix_memalign\(\)*](#) (p. 2033)

The *alignment* argument must be a multiple of `size( void * )` that's also a power of 2.

[*pthread_cond_init\(\)*](#) (p. 2261)

In QNX Neutrino 6.4.0 and later, the *attr* argument is declared as `const pthread_condattr_t *`.

[*pthread_mutex_lock\(\)*](#) (p. 2318)

This function's behavior when you try to lock a mutex that you already own depends on the type of the mutex. For more information, see the entry for [*pthread_mutexattr_settype\(\)*](#) (p. 2357).

[*pthread_mutex_unlock\(\)*](#) (p. 2329)

We've corrected the description of the adjustments to a thread's priority when it unlocks a mutex. If the thread's priority was raised when a higher-priority thread attempted to lock the mutex, the calling thread's priority is again adjusted when it unlocks the mutex; its new priority is the maximum of its own priority and the priorities of those threads it still blocks, either directly or indirectly.

[*pthread_mutexattr_getprotocol\(\)*](#) (p. 2337), [*pthread_mutexattr_setprotocol\(\)*](#) (p. 2351)

PTHREAD_PRIO_NONE was implemented in QNX Neutrino 6.4.0. This protocol means that owning the mutex doesn't affect the priority and scheduling of a thread.

[*putenv\(\)*](#) (p. 2454)

The type of the *env_name* argument is `char *`, not `const char *`.

[*pwrite\(\)*](#) (p. 2470)

We've corrected the list of error codes for this function.

[*random\(\)*](#) (p. 2487), [*srandom\(\)*](#) (p. 3158)

We've corrected some numbers that should have been 2^{nn} , not $2nn$.

[*SchedCtl\(\)*](#) (p. 2745)

We've corrected the name of SCHED_APS_SEC_OFF.

[*seekdir\(\)*](#) (p. 2789)

If you call [*rewinddir\(\)*](#) (p. 2648) on the directory stream, or if you close and reopen the directory, an offset previously returned by [*telldir\(\)*](#) (p. 3375) becomes invalid.

[*sem_open\(\)*](#) (p. 2814)

We've corrected the description of how this function interprets the *name* argument.

[*send\(\)*](#) (p. 2829), [*sendmsg\(\)*](#) (p. 2833), [*sendto\(\)*](#) (p. 2837)

We've corrected the list of error codes for these functions.

[*shm_ctl\(\)*](#) (p. 2925), [*shm_ctl_special\(\)*](#) (p. 2931)

We've corrected the spelling and description of SHMCTL_HIGHUSAGE.

[*shm_open\(\)*](#) (p. 2935)

This function indicates an error of EINVAL if an underlying call to [*resmgr_open_bind\(\)*](#) (p. 2636) fails.

[*signal\(\)*](#) (p. 2974)

Since signal-catching functions are invoked asynchronously with process execution, you need to take into account the same sort of things that you would in a multithreaded program when inspecting or manipulating shared resources; you might have to call [*InterruptLock\(\)*](#) (p. 1174) or [*InterruptUnlock\(\)*](#) (p. 1179), but these functions aren't portable.

If a process sets the action for the SIGCHLD signal to SIG_IGN, its children won't enter the zombie state and the process can't use [*wait\(\)*](#) (p. 3652) or [*waitpid\(\)*](#) (p. 3666) to wait on their deaths.

[*SignalAction\(\)*](#) (p. 2977)

The default action for SIGTSTP, SIGTTIN, and SIGTTOU is to stop the process.

[*sleep\(\)*](#) (p. 3028)

POSIX doesn't define any errors for this function.

[*spawn\(\)*](#) (p. 3089), [*spawnp\(\)*](#) (p. 3122)

The value of *argv[0]* can't be NULL.

[*strptime\(\)*](#) (p. 3233)

We corrected the call to *memset()* in the example.

[*tell64\(\)*](#) (p. 3373)

We've corrected the synopsis of this function.

[*ThreadCreate\(\)*](#) (p. 3402)

To get a pointer to the thread local storage, call *__tls()*.

[*timer_settime\(\)*](#) (p. 3439)

In QNX Neutrino 6.4.0 and later, the *value* argument is of type `const struct itimerspec *`.

[*unsetenv\(\)*](#) (p. 3584)

In QNX Neutrino 6.4.0 and later, this function returns 0 for success, or -1 if an error occurred.

[*wordexp\(\)*](#) (p. 3734)

This function returns *WRDE_NOSYS*; it doesn't set *errno*.

[*write\(\)*](#) (p. 3737)

We've corrected the list of error codes for this function.

What's new in QNX Neutrino 6.4.1?

New entries

[*bt_get_backtrace\(\)*](#) (p. 219)

Collect a backtrace



The backtrace library is an unsupported feature, due to its fragility. For more information, see *Backtraces* in the QNX Neutrino technotes.

[*bt_init_accessor\(\)*](#) (p. 221)

Initialize a backtrace accessor

[*bt_load_memmap\(\)*](#) (p. 223)

Load a memory map associated with a backtrace

[*bt_release_accessor\(\)*](#) (p. 225)

Release an accessor for a backtrace

[*bt_set_flags\(\)*](#) (p. 227)

Set or clear the flags for backtracing

[*bt_sprn_memmap\(\)*](#) (p. 229)

Format the memory map information for a backtrace

[*bt_sprnf_addrs\(\)*](#) (p. 231)

Format the addresses from a backtrace

[*bt_translate_addrs\(\)*](#) (p. 234)

Translate the addresses from a backtrace

[*bt_unload_memmap\(\)*](#) (p. 236)

Unload a memory map associated with a backtrace

[*endfsent\(\)*](#) (p. 495)

Close the filesystem table (/etc/fstab) file

[*fchdir\(\)*](#) (p. 610)

Change the working directory

[*posix_spawnattr_getcred\(\)*](#) (p. 2066)

Get the *uid* and *gid* of the child process

[*getfsent\(\)*](#) (p. 844)

Get the next entry from the filesystem table (/etc/fstab) file

[*getfsfile\(\)*](#) (p. 842)

Search for a filesystem name in the filesystem table (/etc/fstab) file

[*getfsspec\(\)*](#) (p. 847)

Search for a block special device in the filesystem table (/etc/fstab) file

[*getpagesize\(\)*](#) (p. 908)

Get the current page size

[*getpwent_r\(\)*](#) (p. 932)

Get the next entry from the password database

IOFUNC_NOTIFY_DISARM(), *IOFUNC_NOTIFY_INIT()*

Disarm or initialize a notification list; see the entry for [*iofunc_notify\(\)*](#) (p. 1300).

IOFUNC_NOTIFY_INPUT_CHECK(), *IOFUNC_NOTIFY_OBAND_CHECK()*, *IOFUNC_NOTIFY_OUTPUT_CHECK()*

Check a notification list to see if it's worthwhile calling [*iofunc_notify_trigger\(\)*](#) (p. 1308) for it; see the entry for [*iofunc_notify_trigger\(\)*](#).

[*iruserok\(\)*](#) (p. 1419), [*iruserok_sa\(\)*](#) (p. 1419)

Check the identity of a remote host

[*mq_timedreceive_monotonic\(\)*](#) (p. 1722), [*mq_timedsend_monotonic\(\)*](#) (p. 1725)

These functions are similar to [*mq_timedreceive\(\)*](#) and [*mq_timedsend\(\)*](#), but they use `CLOCK_MONOTONIC`, so the timeout isn't affected by changes to the system time.

[*rcmd_af\(\)*](#) (p. 2491)

Execute a command on a remote host

[*resmgr_handle_grow\(\)*](#) (p. 2605)

Expand the capacity of the device manager database

[*resmgr_msgreply\(\)*](#) (p. 2622)

Reply to client with a message

[*resmgr_msgreplyv\(\)*](#) (p. 2625)

Reply to client with a message

[*resmgr_msg_again\(\)*](#) (p. 2616)

Process a message again in a resource manager

[*rresvport_af\(\)*](#) (p. 2670)

Obtain a socket with a privileged address

[*posix_spawnattr_setcred\(\)*](#) (p. 2095)

Set the *uid* and *gid* of the child process

[*setfsent\(\)*](#) (p. 2860)

Open the filesystem table (/etc/fstab) file

[*strlcat\(\)*, *strlcpy\(\)*](#) (p. 3210)

Size-bounded string copying and concatenation

Deprecated content

Instead of using:	Use:
<i>_resmgr_handle_grow()</i>	<i>resmgr_handle_grow()</i> (p. 2605)
<i>_resmgr_io_func()</i>	<i>resmgr_iofuncs()</i> (p. 2614)
<i>_resmgr_ocb()</i>	<i>resmgr_ocb()</i> (p. 2634) (Note that this function takes only one argument, a pointer to a <i>resmgr_context_t</i> structure.)

Changed content

[*atomic_add\(\)*](#) (p. 179), [*atomic_add_value\(\)*](#) (p. 181), [*atomic_clr\(\)*](#) (p. 183), [*atomic_clr_value\(\)*](#) (p. 185), [*atomic_set\(\)*](#) (p. 187), [*atomic_set_value\(\)*](#) (p. 189), [*atomic_sub\(\)*](#) (p. 191), [*atomic_sub_value\(\)*](#) (p. 193), [*atomic_toggle\(\)*](#) (p. 195), [*atomic_toggle_value\(\)*](#) (p. 197)

Perform atomic operations only on objects that were allocated in normal memory mappings. On certain processors, atomic operations will cause a fault if the object is allocated in uncached memory.

[*calloc\(\)*](#) (p. 255), [*malloc\(\)*](#) (p. 1569), [*realloc\(\)*](#) (p. 2527)

The **MALLOC_OPTIONS** environment variable now controls how these functions behave if you specify a size of 0 (or a value of 0 for the *n* argument to *calloc()*). The V (“System V”) and R (“use the *realloc()* behavior of QNX Neutrino 6.4.0 and earlier”) columns below indicate how the functions behave if the value of **MALLOC_OPTIONS** includes that letter:

Function	Default	V	R
<i>calloc(n, 0)</i>	Non-NULL	NULL	No effect
<i>malloc(0)</i>	Non-NULL	NULL	No effect
<i>realloc(NULL, 0)</i>	Non-NULL	NULL	No effect
<i>realloc(non-NULL, 0)</i>	Non-NULL	NULL	NULL

In all the above cases, if the function returns a non-NULL pointer, it's valid only for a corresponding call to [*free\(\)*](#) (p. 717) or *realloc()*.



The default behavior of *realloc(non-NULL, 0)* has changed in this release. Calls like this used to return NULL.

[*ConnectAttach\(\)*](#) (p. 351)

If a process wants other processes to communicate with it, it typically uses [*name_attach\(\)*](#) (p. 1842) to create a channel and associate a name with it, and the sender process uses [*name_open\(\)*](#) (p. 1852) to locate that name and create a connection to it.

[*daemon\(\)*](#) (p. 394)

The High Availability Manager can see death messages only from tasks that are running in session 1, and *daemon()* doesn't put the caller into that session; use [*procmgr_daemon\(\)*](#) (p. 2156) instead if you want to use your application with the HAM.

[*dlopen\(\)*](#) (p. 454)

The description of resolving symbols now includes **LD_PRELOAD**.

[*dlsym\(\)*](#) (p. 461)

This function now supports the special `RTLD_NEXT` flag.

***[fdatsync\(\)](#)* (p. 631), *[fsync\(\)](#)* (p. 787)**

These functions block until the given file has been synchronized. For more information about synchronizing, see “Filesystems and block I/O (devb-*) drivers” in the Fine-Tuning Your System chapter of the QNX Neutrino *User's Guide*.

***[InterruptAttach\(\)](#)* (p. 1147)**

On a multicore system, the interrupt handler runs on the CPU that takes the interrupt.

***[InterruptAttachEvent\(\)](#)* (p. 1154)**

On a multicore system, the thread that receives the event set up by *InterruptAttachEvent()* runs on any CPU, limited only by the scheduler and the runmask.

***[InterruptHookTrace\(\)](#)* (p. 1171)**

The integer that's passed to the handler is a combination of the buffer index and the sequence number; to extract each, pass the integer to the `_TRACE_GET_BUFFNUM()` and `_TRACE_GET_BUFFSEQ()` macros.

***[InterruptWait\(\)](#)* (p. 1183)**

On a multicore system, a thread that calls *InterruptWait()* runs on any CPU, limited only by the scheduler and the runmask.

***[iofunc_notify\(\)](#)* (p. 1300)**

We've added more details about the extended (`_NOTIFY_CONDE_*`) flags.

***[mmap\(\)](#)* (p. 1656)**

- We've documented the following flags:
 - `MAP_NOSYNCFILE`
 - `MAP_BELOW`
 - `MAP_ELF`
 - `MAP_FILE` — for compatibility; has no effect.
 - `MAP_RENAME` — for compatibility; has no effect.
 - `MAP_NORESERVE` — for compatibility; has no effect.
 - `MAP_SYSRAM` — not a valid bit for *mmap()*.

- The *fildev* argument can be the file descriptor for a file, shared memory object, or typed memory object, or NOFD if you're mapping physical memory.
- If you want to map a device's physical memory, use [mmap_device_memory\(\)](#) (p. 1668) instead of *mmap()*; if you want to map a device's registers, use [mmap_device_io\(\)](#) (p. 1666).
- The mapping type controls whether or not changes are propagated to the underlying object. With MAP_PRIVATE, changes aren't propagated; with MAP_SHARED they are.
- You can use MAP_SHARED | MAP_ANON to create a region to be shared by forked applications.
- If you use MAP_PHYS without MAP_ANON, the offset specifies the exact physical address to map (e.g. for video frame buffers). If you use MAP_PHYS with MAP_ANON, *mmap()* allocates physically contiguous memory and ignores the offset.
- We've described how *mmap()* works with typed memory.
- When you're mapping a file, you can specify MAP_FILE, but it isn't necessary. We've added an example of mapping a file.
- You can now create more than one writeable mapping to a file.

[_msg_info](#) (p. 1731)

We've added more information about the *flags* member of this structure.

[MsgDeliverEvent\(\)](#), [MsgDeliverEvent_r\(\)](#) (p. 1736)

These functions can give an error code of EINVAL if the given event isn't valid.

[MsgReceivePulsev\(\)](#) (p. 1768), [MsgReceivePulsev_r\(\)](#) (p. 1768), [MsgReceivev\(\)](#) (p. 1771), [MsgReceivev_r\(\)](#) (p. 1771), [MsgSendsv\(\)](#) (p. 1793), [MsgSendsv_r\(\)](#) (p. 1793), [MsgSendsvnc\(\)](#) (p. 1797), [MsgSendsvnc_r\(\)](#) (p. 1797), [MsgSendv\(\)](#) (p. 1801), [MsgSendv_r\(\)](#) (p. 1801), [MsgSendvnc\(\)](#) (p. 1805), [MsgSendvnc_r\(\)](#) (p. 1805), [MsgSendvs\(\)](#) (p. 1809), [MsgSendvs_r\(\)](#) (p. 1809), [MsgSendvsnc\(\)](#) (p. 1813), [MsgSendvsnc_r\(\)](#) (p. 1813)

These functions now give an error of EOVERFLOW instead of overflowing a buffer if the sum of the user's IOV lengths exceeds INT_MAX.

[name_attach\(\)](#) (p. 1842)

This function can indicate an error of EBUSY if it couldn't create a channel.

[nanospin\(\)](#) (p. 1857), [nanospin_calibrate\(\)](#) (p. 1859), [nanospin_count\(\)](#) (p. 1861), [nanospin_ns\(\)](#) (p. 1863), [nanospin_ns_to_count\(\)](#) (p. 1865)

The *nanospin*()* functions wait for at least the specified time; they might not wait for exactly the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

[*pthread_attr_setstackaddr\(\)*](#) (p. 2229), [*pthread_attr_setstacksize\(\)*](#) (p. 2235)

The system uses some of the provided stack for objects such as thread local storage and an initial stack frame, so less than the entire buffer is available to the thread.

[*pthread_barrier_init\(\)*](#) (p. 2239), [*pthread_cond_init\(\)*](#) (p. 2261), [*pthread_mutex_init\(\)*](#) (p. 2316), [*pthread_once\(\)*](#) (p. 2361), [*pthread_rwlock_init\(\)*](#) (p. 2366), [*sem_init\(\)*](#) (p. 2811) [*SyncTypeCreate\(\)*](#) (p. 3305), [*SyncTypeCreate_r\(\)*](#) (p. 3305)

You should allocate mutexes, condvars, barriers, reader/writer locks, semaphores, and the control structure for *pthread_once()* only in normal memory mappings. On certain processors, atomic operations such as calls to [*pthread_mutex_lock\(\)*](#) (p. 2318) will cause a fault if the control structure is allocated in uncached memory.

[*pthread_cond_timedwait\(\)*](#) (p. 2265)

When setting the time for use with this function, you can use [*nsec2timespec\(\)*](#) (p. 1895) to convert times in nanoseconds to a `timespec` structure (and [*timespec2nsec\(\)*](#) (p. 3474) to convert them back again).

[*resmgr_attach\(\)*](#) (p. 2580)

There's a new flag called `RESMGR_FLAG_CROSS_ENDIAN` that indicates whether the server handles cross-endian support.

The *resmgr_attach()* function copies the *pointers* to the `resmgr_connect_funcs_t` and `resmgr_io_funcs_t` structures, not the structures themselves. You should allocate the structures, declare them to be static, or make them global variables. If your resource manager is for more than one device with different handlers, create separate structures that define the handlers.

[*resmgr_detach\(\)*](#) (p. 2600)

We've described the `_RESMGR_DETACH_CLOSE` flag, which closes all bindings when detaching.

[*setrlimit\(\)*](#) (p. 2899)

In QNX Neutrino, `RLIMIT_DATA` covers all mappings made by the process that are `MAP_ANON` | `MAP_PRIVATE` that aren't `MAP_STACK`, which corresponds typically to heap allocations.

***ThreadCtl(), ThreadCtl_r()* (p. 3408)**

These functions indicate an error of `EINVAL` if the specified runmask is invalid.

***timer_getoverrun()* (p. 3435)**

Note that pulses are queued for a process, so timer overruns occur only for signals.

Errata

***ConnectAttach()* (p. 351)**

You likely need to include `<sys/netmgr.h>` when you use this function, in order to define `ND_LOCAL_NODE`.

***struct dirent* (p. 422)**

In some filesystems, the `d_offset` member identifies the directory entry itself; in others, it's the offset of the `next` directory entry.

***gettimeofday()* (p. 982), *settimeofday()* (p. 2915)**

We've corrected the types of the members of the `timeval` structure.

***InterruptAttachEvent()* (p. 1154)**

An error of `EPERM` indicates that the process doesn't have I/O privileges.

***iofunc_attr_t* (p. 1221)**

The `nlink` member of this structure must be at least 2 for names that represent a directory (one for the directory itself, one for the `./` entry in it).

***iofunc_lock()* (p. 1276)**

This function is actually implemented.

***mq_receive()* (p. 1714)**

If the given `msg_len` is less than the `mq_msgsize` for the given queue, this function sets `errno` to `EMSGSIZE`, not `EINVAL`.

***name_attach()* (p. 1842)**

We've corrected the example so that it now compiles cleanly.

[*resmgr_attach\(\)*](#) (p. 2580)

We've corrected the information about the flags:

- Use the flags whose names *don't* start with an underscore (RESMGR_FLAG_*) in the *flags* member of the *resmgr_attr_t* structure.
- Use the flags whose names *do* start with an underscore (_RESMGR_FLAG_*) in the *flags* argument to this function.

[*resmgr_iofuncs\(\)*](#) (p. 2614)

The public version of this function takes only one argument, a pointer to a *resmgr_context_t* structure.

[*setrlimit\(\)*](#) (p. 2899)

If the RLIMIT_DATA limit is exceeded, the *brk()*, *mmap()*, and *sbrk()* functions fail with *errno* set to ENOMEM.

[*timer_timeout\(\)*](#) (p. 3442), [*TimerTimeout\(\)*](#) (p. 3464)

Because of the nature of time measurement, timers might actually expire later than the specified time. For more information, see the Tick, Tock: Understanding the Microkernel's Concept of Time chapter of the QNX Neutrino *Programmer's Guide*.

[*ttyname\(\)*](#) (p. 3542), [*ttyname_r\(\)*](#) (p. 3544)

We've corrected the list of errors that these functions can indicate.

[*wait\(\)*](#) (p. 3652), [*wait4\(\)*](#) (p. 3658), [*waitid\(\)*](#) (p. 3663), [*waitpid\(\)*](#) (p. 3666)

These functions suspend the calling thread, not the calling process.

What's new in QNX Neutrino 6.4.0?

New entries

[*dirent*](#) (p. 422)

Data structure for a directory entry.

[*dispatch_create_channel\(\)*](#) (p. 436)

Allocate a dispatch handle, specifying a channel ID.

[*insque\(\)*](#) (p. 1145)

Insert an element into a doubly linked queue

[*isfinite\(\)*](#) (p. 1437)

Determine if a number is finite

[*MsgCurrent\(\)*](#) (p. 1734)

Resume processing of a message.

[*posix_spawn\(\)*](#), [*posix_spawnnp\(\)*](#) (p. 2037)

Spawn a process

[*posix_spawn_file_actions_addclose\(\)*](#) (p. 2048)

Add a “close a file descriptor” action to a spawn file actions object

[*posix_spawn_file_actions_adddup2\(\)*](#) (p. 2050)

Add a “duplicate a file descriptor” action to a spawn file actions object

[*posix_spawn_file_actions_addopen\(\)*](#) (p. 2052)

Add an “open a file” action to a spawn file action object

[*posix_spawn_file_actions_destroy\(\)*](#) (p. 2055),

Destroy a spawn file actions object

[*posix_spawn_file_actions_init\(\)*](#) (p. 2057)

Initialize a spawn file actions object

[*posix_spawnattr_addpartid\(\)*](#) (p. 2059)

Specify the ID of a resource partition that the spawned process should be associated with

[*posix_spawnattr_addpartition\(\)*](#) (p. 2061)

Specify the name of a resource partition that the spawned process should be associated with

[*posix_spawnattr_destroy\(\)*](#) (p. 2064)

Destroy a spawn attributes object

[*posix_spawnattr_getcred\(\)*](#) (p. 2066)

Get the user and group ID attributes from a spawn attributes object

[*posix_spawnattr_getflags\(\)*](#) (p. 2068)

Get the POSIX flags from a spawn attributes object

[*posix_spawnattr_getnode\(\)*](#) (p. 2070)

Get the remote node attribute from a spawn attributes object

[*posix_spawnattr_getpartid\(\)*](#) (p. 2072)

Retrieve the partition identifiers currently set in a spawn attributes object

[*posix_spawnattr_getpgroup\(\)*](#) (p. 2074)

Get the process group attribute of a spawn attributes object

[*posix_spawnattr_getrunmask\(\)*](#) (p. 2076)

Get the runmask attribute from a spawn attributes object

[*posix_spawnattr_getschedparam\(\)*](#) (p. 2078)

Get the scheduling parameters attribute of a spawn attributes object

[*posix_spawnattr_getschedpolicy\(\)*](#) (p. 2080)

Get the scheduling policy attribute of a spawn attributes object

[*posix_spawnattr_getsigdefault\(\)*](#) (p. 2082)

Get the “default signals” attribute of a spawn attributes object

[*posix_spawnattr_getsigignore\(\)*](#) (p. 2084)

Get the “ignored signals” attribute from a spawn attributes object

[*posix_spawnattr_getsigmask\(\)*](#) (p. 2086)

Get the signal mask attribute of a spawn attributes object

[*posix_spawnattr_getstackmax\(\)*](#) (p. 2088)

Get the maximum stack size attribute from a spawn attributes object

[*posix_spawnattr_getxflags\(\)*](#) (p. 2090)

Get the POSIX flags and the QNX Neutrino extended flags from a spawn attributes object

[*posix_spawnattr_init\(\)*](#) (p. 2093)

Initialize a spawn attributes object

[*posix_spawnattr_setcred\(\)*](#) (p. 2095)

Set the user and group ID attributes in a spawn attributes object

[*posix_spawnattr_setflags\(\)*](#) (p. 2098)

Set the POSIX flags in a spawn attributes object

[*posix_spawnattr_setnode\(\)*](#) (p. 2101)

Set the remote node attribute in a spawn attributes object

[*posix_spawnattr_setpgroup\(\)*](#) (p. 2103)

Set the process group attribute in a spawn attributes object

[*posix_spawnattr_setrunmask\(\)*](#) (p. 2105)

Set the runmask attribute in a spawn attributes object

[*posix_spawnattr_setschedparam\(\)*](#) (p. 2107)

Set the scheduling parameters attribute of a spawn attributes object

[*posix_spawnattr_setschedpolicy\(\)*](#) (p. 2109)

Set the scheduling policy attribute of a spawn attributes object

[*posix_spawnattr_setsigdefault\(\)*](#) (p. 2111)

Set the “default signals” attribute of a spawn attributes object

[*posix_spawnattr_setsigignore\(\)*](#) (p. 2113)

Set the “ignored signals” attribute of a spawn attributes object

[*posix_spawnattr_setsigmask\(\)*](#) (p. 2115)

Set the signal mask attribute of a spawn attributes object

***posix_spawnattr_setstackmax()* (p. 2117)**

Set the maximum stack size attribute in a spawn attributes object

***posix_spawnattr_setxflags()* (p. 2119)**

Set the POSIX flags and the QNX Neutrino extended flags in a spawn attributes object

***remque()* (p. 2559)**

Remove an element from a doubly linked queue.

RMSK_CLR()*, *RMSK_ISSET()*, *RMSK_SET()*, and *RMSK_SIZE()

You can use these macros to work with the runmask for the `_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT` command for *ThreadCtl()*. For more information, see the entry for *ThreadCtl()* (p. 3408).

***strptime()* (p. 3233)**

Convert a string into a time.

Deprecated content

rsrddbmgr_query()

Use *rsrddbmgr_query_name()* (p. 2693) instead.

sctp_bindx()

Add or remove one or more addresses from a given association.

sctp_connectx()

Help associate an endpoint that is multi-homed.

sctp_freeladdrs()

Free all resources allocated by *sctp_getladdrs()*.

sctp_freepaddrs()

Free all resources allocated by *sctp_getpaddrs()*.

sctp_getladdrs()

Return all locally bound addresses on a socket.

sctp_getpaddrs()

Return all peer addresses in an association.

sctp_peeloff()

Branch off an association into a separate socket.

SCTP

Stream Control Transmission Protocol.

sctp_recvmmsg()

Receive message using advanced SCTP features.

sctp_sendmmsg()

Send message using advanced SCTP features.

Changed content

[*aio_cancel\(\)*](#) (p. 124), [*aio_error\(\)*](#) (p. 127), [*aio_fsync\(\)*](#) (p. 129), [*aio_read\(\)*](#) (p. 131), [*aio_return\(\)*](#) (p. 134), [*aio_suspend\(\)*](#) (p. 136), [*aio_write\(\)*](#) (p. 138), [**aio_cb**](#) (p. 141)

These functions are now fully implemented and documented.

[*_amblksiz*](#) (p. 151)

We've described how you can change the value of this variable.

[*encrypt\(\)*](#) (p. 492)

This function now lets you encrypt or decrypt the given string, depending on the *flags* argument. It isn't safe to call this function from a signal handler or from a multithreaded program.

[*errno*](#) (p. 539)

- There's a new value, EOWNERDEAD, that indicates that the owner of a lock died while holding it.
- The value of EALREADY is changing so that it will be POSIX-compliant; for more information, see “[Changes to EALREADY](#) (p. 547)” in the documentation for *errno*.

[*getaddrinfo\(\)*](#) (p. 818), [*gethostbyname\(\)*](#) (p. 872)

If a name server isn't responding, there's a timeout of 1.5 minutes per name server.

[*getcwd\(\)*](#) (p. 830)

If you call `getcwd( NULL, 0)`, this function allocates a buffer for the name of the directory. POSIX doesn't specify this behavior, but it's common practice.

[*ioctl\(\)*](#) (p. 1198)

We've added a summary of how *ioctl()* commands map to other function calls and [*devctl\(\)*](#) (p. 403) commands.

[*malloc\(\)*](#) (p. 1569)

This entry now describes the environment variables that you can use to control the allocator's behavior.

[*mq_timedreceive\(\)*](#) (p. 1722), [*mq_timedsend\(\)*](#) (p. 1725)

These functions set *errno* to `EINVAL` if the process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

[*procmgr_event_notify\(\)*](#) (p. 2158)

Now describes the `PROC_MGR_EVENT_CONFSTR` and `PROC_MGR_EVENT_SYSCONF` events.

[*pthread_attr_init\(\)*](#) (p. 2213)

We've updated the description of the default thread attributes. The names of the members of the `pthread_attr_t` now start with two underscores (`__`) in order to conform to POSIX.

[*pthread_cond_timedwait\(\)*](#) (p. 2265), [*pthread_cond_wait\(\)*](#) (p. 2269)

In order to conform to POSIX, these functions now return `EPERM` instead of `EINVAL` if the current thread doesn't own the mutex.

[*pthread_mutex_init\(\)*](#) (p. 2316)

If you destroy a mutex by calling *pthread_mutex_destroy()*, don't reuse it without reinitializing it first.

[*pthread_mutex_timedlock\(\)*](#) (p. 2324)

This function now returns `EOWNERDEAD` if the owner of the lock died while holding it.

[*pthread_mutexattr_init\(\)*](#) (p. 2347)

The names of the members of the `pthread_mutexattr_t` now start with two underscores (`__`) in order to conform to POSIX.

[*pulse_attach\(\)*](#) (p. 2442)

We've added more information about the handler function.

[*readdir\(\)*](#) (p. 2514)

A returned value of NULL from this function is ambiguous; if you need to determine if it indicates an error (as opposed to the end of the directory stream), set *errno* to EOK before each call to this function.

[*rsrdbmgr_attach\(\)*](#) (p. 2674)

The description of the *rsrc_request_t* structure now includes the *RSRCDBMGR_FLAG_NAME* and *RSRCDBMGR_FLAG_LIST* flags, and the *name* member.

[*rsrdbmgr_create\(\)*](#) (p. 2681)

The description of the *rsrc_alloc_t* structure now includes the *name* member.

[*shm_ctl\(\)*](#) (p. 2925), [*shm_ctl_special\(\)*](#) (p. 2931)

These functions have a new flag, *SHMCTL_LAZY*. Setting this flag delays allocating memory until it's referenced. If you create anonymous shared memory objects (by calling [*mmap\(\)*](#) (p. 1656) with *MAP_ANON* | *MAP_SHARED* and a file descriptor of -1), a *MAP_LAZY* flag implicitly sets the *SHMCTL_LAZY* flag on the object.

The *shm_ctl_special()* function also has new special flags for ARM platforms.

[*sigevent*](#) (p. 2961)

The thread that receives a *sigevent* of type *SIGEV_SIGNAL_THREAD* depends on the functions associated with the *sigevent*.

[*socketpair\(\)*](#) (p. 3078)

The example of this function now points out that if you write to *fd[0]*, you read from *fd[1]*, and vice versa.

[*spawn\(\)*](#) (p. 3089), [*spawnp\(\)*](#) (p. 3122)

The first three entries (if specified) of the *fd_map* argument become the child process's standard input, standard output, and standard error (*stdin*, *stdout*, and *stderr*).

[*spawn*\(\)*](#)

There's a new `SPAWN_PADDR64_SAFE` flag that permits the selective application spawn of a task that's known to operate safely with 64-bit addressing or doesn't care about the physical memory location.

SyncTypeCreate() (p. 3305)

The names of the members of the `sync_t` now start with two underscores (`__`) in order to conform to POSIX.

sysctl() (p. 3311)

We've added instructions for determining the entries of the *name* array. For a list of commonly used variables, see the entry for `sysctl` in the *Utilities Reference*.

Errata

abort() (p. 54)

We've corrected the description of what this function does.

accept() (p. 58)

This function gives an error of `EINVAL` if you called *accept()* on a socket that you hadn't called *listen()* on.

bind() (p. 208)

This function sets *errno* to `ENOTSOCK` if the given file descriptor isn't for a socket.

brk() (p. 213)

Neutrino doesn't support swapping.

ClockCycles() (p. 325)

It *is* safe to call this function from an interrupt handler.

ConnectAttach()*, *ConnectAttach_r() (p. 351)

These functions can give an error of `ENXIO`.

struct dirent (p. 422)

The *d_name* member is defined as an array of one character; you have to allocate space for the name.

dlopen() (p. 454)

The runtime library search path specified by the `-rpath` option to `ld` is stored in the binary, not in an environment variable.

[GETIOVBASE\(\)](#) (p. 887), [GETIOVLEN\(\)](#) (p. 888)

You need to include `<sys/types.h>`, or else you'll get errors when you compile because `iov_t` isn't defined.

[InterruptAttach\(\)](#) (p. 1147)

The documentation for this function no longer implies that Neutrino supports paging.

[iofunc_fdinfo_default\(\)](#) (p. 1268)

We've corrected the description of what this function returns.

[iofunc_notify\(\)](#) (p. 1300)

This function returns -1 on success to indicate to the resource manager library that it should return a one-part IOV to the client.

[iofunc_openfd\(\)](#) (p. 1327)

This function uses `_IO_FLAG_*` bits, not `O_RDONLY`, etc.

[iofunc_lseek\(\)](#) (p. 1286), [iofunc_lseek_default\(\)](#) (p. 1289)

We've corrected the description of what these functions return.

[iofunc_notify_trigger\(\)](#) (p. 1308)

If the specified *count* is greater than or equal to the trigger count for the particular notification list element, this function calls [MsgDeliverEvent\(\)](#) (p. 1736) to deliver the event to the client.

[malloc\(\)](#) (p. 1569)

We've corrected the description of [_amblksiz](#) (p. 151) in the documentation for `malloc()`.

[mbsrtowcs\(\)](#) (p. 1591), [mbstowcs\(\)](#) (p. 1593)

The *n* argument to these functions is the number of wide characters, not the number of bytes.

[mem_offset\(\)](#) (p. 1600)

This function gives an error of `E_OVERFLOW` (not `E2BIG`) if the address is too large for the 32-bit *off_t*.

[mq_send\(\)](#) (p. 1717)

If the number of elements on the specified queue is equal to its *mq_maxmsg*, and *O_NONBLOCK* *wasn't* set (in the *oflag* argument to *mq_open()*), the call to *mq_send()* blocks.

[msync\(\)](#) (p. 1825)

- We've corrected the description of the *MS_CACHE_ONLY* flag (a QNX Neutrino extension).
- This function sets *errno* to *EINTR* if the call is interrupted by a signal.

[name_attach\(\)](#) (p. 1842)

- This function fails and sets *errno* to *EEXIST* if the specified path already exists.
- If *name_attach()* creates a channel, the channel will have the *_NTO_CHF_DISCONNECT*, *_NTO_CHF_COID_DISCONNECT* and *_NTO_CHF_UNBLOCK* flags set.

[name_close\(\)](#) (p. 1848)

This function operates on a side-channel connection ID, not a file descriptor.

[name_open\(\)](#) (p. 1852)

This function doesn't attach to a name; it searches for the given name.

[pathmgr_symlink\(\)](#) (p. 1943)

The order of the arguments is now correct.

[posix_mem_offset\(\)](#), [posix_mem_offset64\(\)](#) (p. 2030)

We corrected the description of the *contig_len* argument.

[posix_typed_mem_open\(\)](#) (p. 2124)

The *tflag* argument is of type *int*.

We've corrected the details about the resolution of the name of the typed memory object.

[pthread_barrier_wait\(\)](#) (p. 2241)

This function returns *PTHREAD_BARRIER_SERIAL_THREAD* to one of the threads waiting at a barrier.

[pthread_cond_timedwait\(\)](#) (p. 2265)

We've corrected the example so that it runs correctly for timeouts of less than 1 second.

[*pthread_getname_np\(\)*](#) (p. 2297), [*pthread_setname_np\(\)*](#) (p. 2398)

You don't have to be root in order to get or set a thread's name.

[*pthread_mutexattr_init\(\)*](#) (p. 2347)

We corrected the synopsis of this function.

[*pthread_mutex_lock\(\)*](#) (p. 2318)

You can allow recursive behavior by using [*pthread_mutexattr_setrecursive\(\)*](#) (p. 2355) to set the attribute to PTHREAD_RECURSIVE_ENABLE after calling [*pthread_mutex_init\(\)*](#) (p. 2316).

[*readdir_r\(\)*](#) (p. 2518)

This function doesn't set *errno*; it returns any error codes.

[*resmgr_context_t*](#) (p. 2598)

The *info* member of this structure is a `_msg_info` structure, not a pointer to one.

[*resmgr_handle_tune\(\)*](#) (p. 2606)

This function is declared in `<sys/resmgr.h>`.

[*resmgr_io_funcs_t*](#) (p. 2610)

We corrected the name and definition of the *opened* member.

[*resmgr_iofuncs\(\)*](#) (p. 2614)

This function is declared in `<sys/resmgr.h>`, not `<resmgr.h>`.

[*_RESMGR_NPARTS\(\)*](#) (p. 2632)

This macro returns the negative of the number of parts (to distinguish the number from error codes that the resource manager's handler functions might return).

[*sbrk\(\)*](#) (p. 2702)

Neutrino doesn't support swapping.

[*SETIOV\(\)*](#) (p. 2870)

You need to include `<sys/types.h>`, or else you'll get errors when you compile because `iov_t` isn't defined.

***shm_open()* (p. 2935)**

You can't open shared memory across the network.

***sigprocmask()* (p. 3004)**

The classification for this function is POSIX 1003.1 CX.

***snprintf()* (p. 3070), *sprintf()* (p. 3151), *sscanf()* (p. 3162), *swprintf()* (p. 3272), *wscanf()* (p. 3274)**

It's safe to call these functions in a signal handler if the data isn't floating point.

***socket()* (p. 3075)**

This function sets *errno* to EAFNOSUPPORT if the specified address family isn't supported.

***spawn()* (p. 3089)**

We've corrected the description of the SPAWN_CHECK_SCRIPT flag in the inheritance structure.

***SYSPAGE_ENTRY()* (p. 3326)**

The *sec* variable in the example is now a *double* instead of a *float*.

***timer_create()* (p. 3427)**

We've corrected the description of CLOCK_SOFTTIME.

***TimerTimeout()* (p. 3464)**

The timeout value *starts timing out* when *TimerTimeout()* is called, *not* when the blocking state is entered. It might be possible to get preempted after calling *TimerTimeout()* but before the blocking kernel call.

***wcstol()* (p. 3709), *wcstoll()* (p. 3709), *wcstoul()* (p. 3713), *wcstoull()* (p. 3713)**

These functions are declared in *<wchar.h>*, not *<stdlib.h>*.

What's new in QNX Neutrino 6.3.2?

Errata

clock() (p. 305)

This function returns the number of clock ticks, or `(clock_t) -1` if the number of ticks couldn't be determined or exceeds the maximum value that the `clock_t` type can represent.

GETIOVBASE() (p. 887), *GETIOVLEN()* (p. 888)

The argument to these macros is a pointer to an `iov_t` structure.

What's new in the QNX Neutrino Core OS 6.3.2?

New entries

[*posix_typed_mem_get_info\(\)*](#) (p. 2122), [*posix_typed_mem_open\(\)*](#) (p. 2124)

New functions for POSIX support of memory-mapping.

[*pthread_getname_np\(\)*](#) (p. 2297), [*pthread_setname_np\(\)*](#) (p. 2398)

The *pthread_getname_np()* function retrieves the name of a thread. The *pthread_setname_np()* function names a thread in your application.

[*pthread_attr_getstackprealloc\(\)*](#) (p. 2209), [*pthread_attr_setstackprealloc\(\)*](#) (p. 2233)

Get and set the memory to preallocate for a MAP_LAZY stack.

[*munmap_flags\(\)*](#) (p. 1838)

New function to unmap previously mapped addresses exercising more control.

[*SchedCtl\(\)*](#) (p. 2745)

Control the adaptive partitioning scheduler

[*SIGEV_CLEAR_CRITICAL\(\)*](#)

In adaptive partitioning, make a thread run as noncritical. See [sigevent](#) (p. 2961).

[*SIGEV_GET_TYPE\(\)*](#)

Determine the type of a sigevent event. See [sigevent](#) (p. 2961).

[*SIGEV_MAKE_CRITICAL\(\)*](#)

In adaptive partitioning, make a thread run as critical. See [sigevent](#) (p. 2961).

Changed content

[*ChannelCreate\(\)*](#) (p. 278)

If you're using adaptive partitioning and you set `_NTO_CHF_FIXED_PRIORITY`, the receiving thread won't run in the sending thread's partition.

[*getrlimit\(\)*](#) (p. 942), [*getrlimit64\(\)*](#) (p. 942), [*setrlimit\(\)*](#) (p. 2899), [*setrlimit64\(\)*](#) (p. 2899)

Added the following resources:

- RLIMIT_MEMLOCK
- RLIMIT_NPROC
- RLIMIT_NTHR
- RLIMIT_OFILE
- RLIMIT_RSS

[*iofunc_open_default\(\)*](#) (p. 1325)

The documentation now describes the *extra* argument.

[*mlock\(\)*](#) (p. 1651), [*mlockall\(\)*](#) (p. 1653), [*munlock\(\)*](#) (p. 1828)

The documentation now describes the POSIX support for memory mapping.

[*mmap\(\)*](#) (p. 1656)

The documentation now includes a new MAP_NOINIT flag.

[*msync\(\)*](#) (p. 1825)

The documentation now describes new MS_CACHE_ONLY flag.

[*posix_mem_offset\(\)*](#) (p. 2030), [*posix_mem_offset64\(\)*](#) (p. 2030)

The documentation now describes the POSIX support for memory-mapping.

[*sigevent*](#) (p. 2961)

In the future, the library might use hidden bits in the *sigev_notify* member of the *sigevent* structure. Don't compare this member directly to values such as SIGEV_PULSE; use the *SIGEV_GET_TYPE()* macro.

[*spawn\(\)*](#) (p. 3089)

The documentation now describes the SPAWN_EXPLICIT_CPU flag and the *runmask* member of the inheritance structure.

[*ThreadCtl\(\)*](#) (p. 3408)

The documentation now contains the following commands:

- _NTO_TCTL_NAME
- _NTO_TCTL_ONE_THREAD_HOLD
- _NTO_TCTL_ONE_THREAD_CONT
- _NTO_TCTL_RUNMASK_GET_AND_SET
- _NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT

What's new in QNX Neutrino 6.3.0 Service Pack 2?

New entries

[*cache_fini\(\)*](#) (p. 244)

Free cache-coherency resources when the driver is unloaded.

[*CACHE_FLUSH\(\)*](#) (p. 246)

Flush cache line associated to a data buffer.

[*cache_init\(\)*](#) (p. 248)

Register with the cache coherency library.

[*CACHE_INVALID\(\)*](#) (p. 253)

Invalidate cache line associated to a data buffer.

Changed content

[*mq_close\(\)*](#) (p. 1702), [*mq_getattr\(\)*](#) (p. 1704), [*mq_notify\(\)*](#) (p. 1707), [*mq_open\(\)*](#) (p. 1709), [*mq_receive\(\)*](#) (p. 1714), [*mq_send\(\)*](#) (p. 1717), [*mq_setattr\(\)*](#) (p. 1720), [*mq_timedreceive\(\)*](#) (p. 1722), [*mq_timedsend\(\)*](#) (p. 1725), [*mq_unlink\(\)*](#) (p. 1728)

Added information about the traditional (mqueue) and alternate (mq) implementations of message queues.

[*nanospin\(\)*](#) (p. 1857), [*nanospin_calibrate\(\)*](#) (p. 1859), [*nanospin_count\(\)*](#) (p. 1861), [*nanospin_ns\(\)*](#) (p. 1863), [*nanospin_ns_to_count\(\)*](#) (p. 1865)

The *nanospin*()* functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX [*timer_**](#) (p. 3427) functions.

[*pci_attach_device\(\)*](#) (p. 1966), [*pci_find_class\(\)*](#) (p. 1979), [*pci_find_device\(\)*](#) (p. 1981)

For a list of supported device and vendor IDs, see <hw/pci_devices.h>; for a list of class and subclass codes, see <hw/pci.h>.

Errata

[*_cmdname\(\)*](#) (p. 342)

Corrected the information about what this function returns.

[*openlog\(\)*](#) (p. 1918)

Corrected the name of the LOG_FTP facility.

[*pci_attach_device\(\)*](#) (p. 1966)

The *BusNumber* and *DevFunc* members of the *pci_dev_info* structure are input/output.

[*pci_irq_routing_options\(\)*](#) (p. 1983)

This function is for x86 only.

[*sem_close\(\)*](#) (p. 2805), [*sem_open\(\)*](#) (p. 2814), [*sem_unlink\(\)*](#) (p. 2825)

Named semaphores are now managed by *procnto*, not *mqueue*.

[*thread_pool_create\(\)*](#) (p. 3387)

The description of the *block_func* member of the *thread_pool_attr_t* structure has been corrected.

[*vfprintf\(\)*](#) (p. 3616), [*vprintf\(\)*](#) (p. 3624)

The code samples now compile without warnings.

What's new in QNX Neutrino 6.3.0 Service Pack 1?

New entries

[*shm_ctl_special\(\)*](#) (p. 2931)

Give special attributes to a shared memory object

Changed content

[*getdomainname\(\)*](#) (p. 833)

If the buffer isn't large enough, *getdomainname()* truncates the domain name.

[*getgrouplist\(\)*](#) (p. 861)

The Neutrino implementation of this function ignores the *basegid* argument.

[*mallopt\(\)*](#) (p. 1575)

The MALLOC_CHKACCESS, MALLOC_FILLAREA, and MALLOC_CKCHAIN options were added to this call.

[*MsgReply\(\)*](#) (p. 1775), [*MsgReply_r\(\)*](#) (p. 1775), [*MsgReplyv\(\)*](#) (p. 1778), [*MsgReplyv_r\(\)*](#) (p. 1778)

The *MsgSend*_r()* functions use negative *errno* values to indicate failure, so you shouldn't pass a negative value for the *status* to *MsgReply*()*, because the *MsgSend*_r()* functions could interpret it as an error code.

[*name_attach\(\)*](#) (p. 1842)

The example now handles an *_IO_CONNECT* message.

[*nanospin\(\)*](#) (p. 1857), [*nanospin_ns\(\)*](#) (p. 1863), and [*nanospin_ns_to_count\(\)*](#) (p. 1865)

The first time that you call these functions, the C library invokes [*nanospin_calibrate\(\)*](#) (p. 1859) with an argument of 0 (interrupts enabled), unless you call it directly first.

[*pci_attach_device\(\)*](#) (p. 1966)

Added PCI_MASTER_ENABLE to the flags.

[*procmgr_daemon\(\)*](#) (p. 2156)

The data in the `siginfo_t` structure for the SIGCHLD signal that the parent receives isn't useful in this case.

[rsrddbmgr_create\(\)](#) (p. 2681)

Added RSRCDBMGR_FLAG_NOREMOVE to the flags.

[setenv\(\)](#) (p. 2853)

This function doesn't free any memory. If you want to change the value of an existing environment variable, you should use [putenv\(\)](#) (p. 2454) instead.

[sigevent](#) (p. 2961)

If you don't want to modify the priority of the thread that receives the pulse, specify SIGEV_PULSE_PRIO_INHERIT for the *priority* when you call `SIGEV_PULSE_INIT()`.

[spawn\(\)](#) (p. 3089), [spawnp\(\)](#) (p. 3122)

- Added descriptions of the rest of the flags for the `inheritance` structure.
- If you set SPAWN_EXEC in the *flags* member of the `inheritance` structure, these functions don't return, unless an error occurred.

Errata

[getsubopt\(\)](#) (p. 977)

Corrected the example.

[mq_notify\(\)](#) (p. 1707)

Don't use SIGEV_INTR as an event type.

[pause\(\)](#) (p. 1947)

This function suspends the calling thread, not the process, until delivery of a signal.

[pthread_setschedparam\(\)](#) (p. 2400)

- The timeslice for round-robin scheduling (SCHED_RR) is 4 × the clock period.
- You can specify sporadic scheduling at any time, not just when you create a thread.

regexec() (p. 2549)

Corrected the description of the `regmatch_t` structure.

sched_get_priority_max() (p. 2725), ***sched_get_priority_min()*** (p. 2727), ***sched_setscheduler()*** (p. 2740), ***SchedInfo()*** (p. 2777), ***SchedSet()*** (p. 2780)

The timeslice for round-robin scheduling (`SCHED_RR`) is $4 \times$ the clock period.

sched_setscheduler() (p. 2740), ***SchedSet()*** (p. 2780)

You can specify sporadic scheduling at any time, not just when you create a thread.

spawn() (p. 3089), ***spawnl()*** (p. 3099), ***spawnle()*** (p. 3105), ***spawnlp()*** (p. 3111), ***spawnlpe()*** (p. 3116), ***spawnnp()*** (p. 3122), ***spawnv()*** (p. 3129), ***spawnve()*** (p. 3134), ***spawnvp()*** (p. 3140), ***spawnvpe()*** (p. 3145)

The child process can't access the parent process's environment, only its own.

What's new in QNX Neutrino 6.3.0?

New entries

[*fopen64\(\)*](#) (p. 681)

Large-file support for *fopen()*.

[*freopen64\(\)*](#) (p. 721)

Large-file support for *freopen()*.

[*ftw64\(\)*](#) (p. 798)

Large-file support for *ftw()*.

[*getnameinfo\(\)*](#) (p. 894)

Perform address-to-nodename translation.

[*inet6_option_alloc\(\)*](#) (p. 1103)

Append IPv6 hop-by-hop or destination options to an ancillary data object

[*inet6_option_append\(\)*](#) (p. 1105)

Append an IPv6 hop-by-hop or destination option to an ancillary data object

[*inet6_option_find\(\)*](#) (p. 1107)

Search for IPv6 hop-by-hop and destination options

[*inet6_option_init\(\)*](#) (p. 1109)

Initialize an ancillary data object that contains IPv6 hop-by-hop and destination options

[*inet6_option_next\(\)*](#) (p. 1111)

Find the next IPv6 hop-by-hop or destination option

[*inet6_option_space\(\)*](#) (p. 1113)

Determine how much space an IPv6 hop-by-hop or destination option requires

[*inet6_rthdr_add\(\)*](#) (p. 1115)

Add an address to an IPv6 routing header

[*inet6_rthdr_getaddr\(\)*](#) (p. 1117)

Get a pointer to an IPv6 address in the routing header

[*inet6_rthdr_getflags\(\)*](#) (p. 1119)

Get the flags for a segment in an IPv6 routing header

[*inet6_rthdr_init\(\)*](#) (p. 1121)

Initialize an IPv6 routing header

[*inet6_rthdr_lasthop\(\)*](#) (p. 1123)

Specify the Strict/Loose flag for the final hop of an IPv6 routing header

[*inet6_rthdr_reverse\(\)*](#) (p. 1125)

Reverse the list of addresses in an IPv6 router header

[*inet6_rthdr_segments\(\)*](#) (p. 1127)

Count the segments in an IPv6 routing header

[*inet6_rthdr_space\(\)*](#) (p. 1129)

Determine the space required by an IPv6 routing header

[*ipsec_dump_policy\(\)*](#) (p. 1402)

Generate a readable string from an IPsec policy specification.

[*ipsec_get_policylen\(\)*](#) (p. 1404)

Get the length of the IPsec policy.

[*ipsec_set_policy\(\)*](#) (p. 1406)

Generate an IPsec policy specification structure from a readable string.

[*nftw\(\)*](#) (p. 1888), [*nftw64\(\)*](#) (p. 1888)

Walk a file tree and its large-file support.

[*poll\(\)*](#) (p. 2013)

Input/output multiplexing.

[*resmgr_handle_tune\(\)*](#) (p. 2606)

Tune aspects of client *fd*-to-OCB mapping

[*sctp_bindx\(\)*](#)

Add or remove one or more addresses from a given association.

sctp_connectx()

Help associate an endpoint that is multi-homed.

sctp_freeladdrs()

Free all resources allocated by *sctp_getladdrs()*.

sctp_freepaddrs()

Free all resources allocated by *sctp_getpaddrs()*.

sctp_getladdrs()

Return all locally bound addresses on a socket.

sctp_getpaddrs()

Return all peer addresses in an association.

sctp_peeloff()

Branch off an association into a separate socket.

SCTP

Stream Control Transmission Protocol.

sctp_recvmmsg()

Receive message using advanced SCTP features.

sctp_sendmmsg()

Send message using advanced SCTP features.

[tmpfile64\(\)](#) (p. 3478)

Large-file support for *tmpfile()*.

Deprecated content

Instead of:	Use:
<i>drem()</i>	remainder() (p. 2554)
<i>dremf()</i>	remainder() (p. 2554)
<i>finite()</i>	isfinite() (p. 1437)
<i>finitef()</i>	isfinite() (p. 1437)
<i>isinf()</i>	isinf() (p. 1441)

Instead of:	Use:
<i>isnanf()</i>	<i>isnan()</i> (p. 1445)
<i>significand()</i>	<i>scalbn</i> (p. 2707)(<i>x</i> , <i>-ilogb</i> (p. 1057)(<i>x</i>))
<i>significandf()</i>	<i>scalbnf</i> (p. 2707)(<i>x</i> , <i>-ilogbf</i> (p. 1057)(<i>x</i>))
<i>scalbf()</i>	<i>scalbn()</i> (p. 2707)

What's new in QNX Neutrino 6.2.1?

New entries

[*dispatch_unblock\(\)*](#) (p. 445)

Unblock all of the threads that are blocked on a dispatch handle

[*errno*](#) (p. 539)

Each thread in a multi-threaded program has its own error value in its thread local storage. No matter which thread you're in, you can simply refer to *errno* — it's defined in such a way that it refers to the correct variable for the thread. For more information, see “[Local storage for private data](#) (p. 3405)” in the documentation for *ThreadCreate()*.

[*pthread_attr_setschedpolicy\(\)*](#) (p. 2223).

Sporadic scheduling (SCHED_SPORADIC) is a new feature of QNX Neutrino 6.2.0.

[*sched_param*](#) (p. 2731)

Structure of scheduling parameters

[*va_copy\(\)*](#) (p. 3605)

Make a copy of a variable argument list

Changed content

[*bind\(\)*](#) (p. 208), [*bindresvport\(\)*](#) (p. 211)

These functions aren't cancellation points any more, because this conflicted with POSIX.

[*htonl\(\)*](#) (p. 1026), [*htons\(\)*](#) (p. 1027), [*inet_ntop\(\)*](#) (p. 1093), [*inet_pton\(\)*](#) (p. 1096), [*isfdtype\(\)*](#) (p. 1435), [*ntohl\(\)*](#) (p. 1896), [*ntohs\(\)*](#) (p. 1897)

These functions have been moved from libsocket to libc.

Errata

[*alphasort\(\)*](#) (p. 149)

This function compares two directory entries; it doesn't sort an array of entries.

[*execipe\(\)*](#) (p. 568), [*execvpe\(\)*](#) (p. 591)

You can now execute a shell script.

[*fgetc\(\)*](#) (p. 645), [*fgetchar\(\)*](#) (p. 648), [*fgets\(\)*](#) (p. 652), [*fgetwc\(\)*](#) (p. 656), [*fgetws\(\)*](#) (p. 658), [*getc\(\)*](#) (p. 823), [*getc_unlocked\(\)*](#) (p. 825), [*getchar\(\)*](#) (p. 827), [*getchar_unlocked\(\)*](#) (p. 829), [*gets\(\)*](#) (p. 950), [*getw\(\)*](#) (p. 991), [*getwc\(\)*](#) (p. 993), [*getwchar\(\)*](#) (p. 995)

Use [*feof\(\)*](#) (p. 637) or [*ferror\(\)*](#) (p. 639) to distinguish an end-of-file condition from an error.

[*fstat\(\)*](#), [*fstat64\(\)*](#) (p. 779)

These functions return -1 if an error occurs.

[*iofunc_mmap\(\)*](#) (p. 1294), [*iofunc_mmap_default\(\)*](#) (p. 1298)

These functions return a nonpositive value on success.

[*InterruptAttach\(\)*](#) (p. 1147), [*InterruptAttachEvent\(\)*](#) (p. 1154)

You should always set `_NTO_INTR_FLAGS_TRK_MSK`.

[*mq_getattr\(\)*](#) (p. 1704), [*mq_setattr\(\)*](#) (p. 1720)

The *mq_flags* member of the `mq_attr` structure applies to the message-queue description (i.e. locally), not to the queue as a whole.

[*mq_open\(\)*](#) (p. 1709)

Corrected the interpretation of the *name* argument.

[*MsgError\(\)*](#), [*MsgError_r\(\)*](#) (p. 1743)

If the *error* argument is EOK, the *MsgSend*()* call returns EOK; if *error* is any other value, the *MsgSend*()* call returns -1.

[*MsgSendPulse\(\)*](#), [*MsgSendPulse_r\(\)*](#) (p. 1789)

You can now send pulses across the network.

You can send a pulse to any process — not just to a process in the same process group — if your process has the appropriate permission.

[*name_open\(\)*](#) (p. 1852)

This function returns a nonnegative integer representing a side-channel connection ID, or -1 if an error occurred.

[*printf\(\)*](#) (p. 2133)

The exponent produced for the e and E formats is at least two digits long.

Clarified what happens if the format string includes invalid multibyte characters.

[*pthread_mutex_timedlock\(\)*](#) (p. 2324), [*pthread_rwlock_timedrdlock\(\)*](#) (p. 2371),
[*pthread_rwlock_timedwrlock\(\)*](#) (p. 2373)

The timeout is based on the CLOCK_REALTIME clock.

[*_resmgr_ocb\(\)*](#)

Corrected the name; see [*resmgr_ocb\(\)*](#) (p. 2634).

[*select\(\)*](#) (p. 2791)

This function and the associated macros are now defined in `<sys/select.h>`, instead of `<sys/time.h>` (which includes `<sys/select.h>`).

[*sem_open\(\)*](#) (p. 2814)

Corrected the interpretation of the *sem_name* argument.

[*sem_timedwait\(\)*](#) (p. 2820)

The timeout is based on the CLOCK_REALTIME clock.

[*send\(\)*](#) (p. 2829)

The list of errors now includes EPIPE.

[*shm_open\(\)*](#) (p. 2935)

Corrected the interpretation of the *name* argument.

[*sigaction\(\)*](#) (p. 2948)

Corrected the example (it isn't safe to call [*printf\(\)*](#) (p. 2133) in a signal handler).

[*spawn\(\)*](#) (p. 3089), [*spawnl\(\)*](#) (p. 3099), [*spawnle\(\)*](#) (p. 3105), [*spawnlp\(\)*](#) (p. 3111), [*spawnlpe\(\)*](#) (p. 3116),
[*spawnnp\(\)*](#) (p. 3122), [*spawnvp\(\)*](#) (p. 3129), [*spawnve\(\)*](#) (p. 3134), [*spawnvpe\(\)*](#) (p. 3140), [*spawnvpe\(\)*](#) (p. 3145)

You can now execute a shell script.

The child process's *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are now calculated.

[*timer_create\(\)*](#) (p. 3427)

Don't use SIGEV_INTR or SIGEV_UNBLOCK for the event type.

[*vsnprintf\(\)*](#) (p. 3632)

Corrected the returned values.

What's new in QNX Neutrino 6.2?

New entries

addrinfo (p. 122)

TCP/IP address information

dircntl() (p. 419)

Control an open directory

freeaddrinfo() (p. 719)

Free an address information structure

freeifaddrs() (p. 720)

Free an address information structure

gai_strerror() (p. 814)

Return the *getaddrinfo()* error code

getaddrinfo() (p. 818)

Get address information

getdomainname() (p. 833)

Get the domain name of the current host

gethostbyname2() (p. 872)

Get a network host entry, given a name

getifaddrs() (p. 885)

Get a network interface address

hwi_find_item() (p. 1028)

Find an item in the *hwi_item* structure

hwi_find_tag() (p. 1030)

Find a tag in the *hwi_item* structure

hwi_off2tag() (p. 1037)

Return a pointer to the start of a tag in the *hwinfo* area of the system page

***hwi_tag2off()* (p. 1039)**

Return the offset from the start of the *hwinfo* area of the system page

***ICMP6* (p. 1047)**

Internet Control Message Protocol for IPv6

***if_freenameindex()* (p. 1049)**

Free dynamic memory allocated by *if_nameindex()*

***if_indextoname()* (p. 1050)**

Map an interface index to its name

***if_nameindex()* (p. 1052)**

Return a list of interfaces

***if_nametoindex()* (p. 1054)**

Map an interface name to its index

***ifaddrs()* (p. 1055)**

Structure that describes an Internet host

***INET6* (p. 1100)**

Internet Protocol version 6 family

***inet_net_ntop()* (p. 1080)**

Convert an Internet network number to CIDR format

***inet_net_pton()* (p. 1085)**

Convert an Internet network number from CIDR format to network format

***IPv6* (p. 1412)**

Internet Protocol version 6

***IPsec* (p. 1396)**

Internet security protocol

***mallinfo()* (p. 1568)**

Get memory allocation information

***mallopt()* (p. 1575)**

Control the memory allocation

[*mcheck\(\)*](#) (p. 1598)

Enable memory allocation routine consistency checks

[*memalign\(\)*](#) (p. 1603)

Allocate aligned memory

[*mprobe\(\)*](#) (p. 1697)

Perform consistency check on memory

[*posix_memalign\(\)*](#) (p. 2033)

Allocate aligned memory

[*procmgr_session\(\)*](#) (p. 2175)

Provide process manager session support

[*_resmgr_handle_grow\(\)*](#)

Expand the capacity of the device manager database; see
[*resmgr_handle_grow\(\)*](#) (p. 2605)

[*_resmgr_io_func\(\)*](#)

Retrieve an I/O function from an I/O function table

[*resmgr_iofuncs\(\)*](#) (p. 2614)

Extract the I/O function pointers associated with client connections

[*_resmgr_ocb\(\)*](#)

Retrieve an Open Control Block; see [*resmgr_ocb\(\)*](#) (p. 2634).

[*sched_get_priority_adjust\(\)*](#) (p. 2723)

Calculate the allowable priority for the scheduling policy

[*seekdir\(\)*](#) (p. 2789)

Set the position for the next read of the directory stream

[*_sleepon_broadcast\(\)*](#) (p. 3030)

Wake up multiple threads

[*_sleepon_destroy\(\)*](#) (p. 3032)

Destroy a sleepon

[***_sleepon_init\(\)***](#) (p. 3034)

Initialize a sleepon

[***_sleepon_lock\(\)***](#) (p. 3036)

Lock a sleepon

[***_sleepon_signal\(\)***](#) (p. 3038)

Wake up a single thread

[***_sleepon_unlock\(\)***](#) (p. 3040)

Unlock a sleepon

[***_sleepon_wait\(\)***](#) (p. 3042)

Wait on a sleepon

[***tcsetsid\(\)***](#) (p. 3369)

Make a terminal device a controlling device

[***strtoimax\(\), strtoumax\(\)***](#) (p. 3256)

Convert a string to an integer type

[***telldir\(\)***](#) (p. 3375)

Get the location associated with the directory stream

[***valloc\(\)***](#) (p. 3610)

Allocate a heap block aligned on a page boundary

[***wcstoimax\(\), wcstoumax\(\)***](#) (p. 3705)

Convert a wide-character string to an integer type

Deprecated content

- *getpriority()* — use [*getprio\(\)*](#) (p. 922) or [*SchedGet\(\)*](#) (p. 2774) instead.
- *setpriority()* — use [*setprio\(\)*](#) (p. 2890) or [*SchedSet\(\)*](#) (p. 2780) instead.

Errata

[***snprintf\(\)***](#) (p. 3070)

Corrected the Returns section and Classifications

What's new in QNX Neutrino 6.1.0?

New entries

The following functions have been added:

Wide-character functions (p. 3651)

Wide-character versions of many functions

InterruptHookTrace() (p. 1171)

Attach the pseudo interrupt handler that's used by the instrumented module

iofdinfo() (p. 1210)

Retrieve server attributes

iofunc_finfo() (p. 1266)

Handle an `_IO_FDINFO` message

iofunc_finfo_default() (p. 1268)

Default handler for `_IO_FDINFO` messages

MsgVerifyEvent(), MsgVerifyEvent_r() (p. 1817)

Check the validity of a receive ID and an event configuration

resmgr_unbind() (p. 2644)

Remove an OCB

straddstr() (p. 3178)

Concatenate one string on to the end of another

SyncCtl(), SyncCtl_r() (p. 3286)

Perform an operation on a synchronization object

SyncMutexEvent(), SyncMutexEvent_r() (p. 3291)

Attach an event to a mutex

SyncMutexRevive(), SyncMutexRevive_r() (p. 3297)

Revive a mutex

thread_pool_control() (p. 3385)

Control the thread pool behavior

[*thread_pool_limits\(\)*](#) (p. 3395)

Wrapper function for *thread_pool_control()*

[*TraceEvent\(\)*](#) (p. 3510)

Trace kernel events

Deprecated content

[*matherr\(\)*](#)

Handle errors in math library functions

A20 gate

On x86-based systems, a hardware component that forces the A20 address line on the bus to zero, regardless of the actual setting of the A20 address line on the processor. This component is in place to support legacy systems, but the QNX Neutrino RTOS doesn't require any such hardware. Note that some processors, such as the 386EX, have the A20 gate hardware built right into the processor itself — our IPL will disable the A20 gate as soon as possible after startup.

adaptive

Scheduling policy whereby a thread's priority is decayed by 1. See also *FIFO*, *round robin*, and *sporadic*.

adaptive partitioning

A method of dividing, in a flexible manner, CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

application ID

A number that identifies all processes that are part of an application. Like process group IDs, the application ID value is the same as the process id of the first process in the application. A new application is created by spawning with the `POSIX_SPAWN_NEWAPP` or `SPAWN_NEWAPP` flag. A process created without one of those inherits the application ID of its parent. A process needs the `PROCMGR_AID_CHILD_NEWAPP` ability in order to set those flags.

The *SignalKill()* kernel call accepts a `SIG_APPID` flag ORed into the signal number parameter. This tells it to send the signal to all the processes with an application ID that matches the *pid* argument. The `DCMD_PROC_INFO devctl()` returns the application ID in a structure field.

asymmetric multiprocessing (AMP)

A multiprocessing system where a separate OS, or a separate instantiation of the same OS, runs on each CPU.

atomic

Of or relating to atoms. :-)

In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered *indivisible*. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another

thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread's position.

attributes structure

Structure containing information used on a per-resource basis (as opposed to the *OCB*, which is used on a per-open basis).

This structure is also known as a *handle*. The structure definition is fixed (`iofunc_attr_t`), but may be extended. See also *mount structure*.

bank-switched

A term indicating that a certain memory component (usually the device holding an *image*) isn't entirely addressable by the processor. In this case, a hardware component manifests a small portion (or “window”) of the device onto the processor's address bus. Special commands have to be issued to the hardware to move the window to different locations in the device. See also *linearly mapped*.

base layer calls

Convenient set of library calls for writing resource managers. These calls all start with *resmgr_**(). Note that while some base layer calls are unavoidable (e.g. *resmgr_pathname_attach()*), we recommend that you use the *POSIX layer calls* where possible.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an “extension” to the BIOS or ROM Monitor — control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes 0x55 and 0xAA must be present (in that order) as the first two bytes in the device, with control being transferred to offset 0x0003.

block-integral

The requirement that data be transferred such that individual structure components are transferred in their entirety — no partial structure component transfers are allowed.

In a resource manager, directory data must be returned to a client as *block-integral* data. This means that only complete `struct dirent` structures can be returned — it's inappropriate to return partial structures,

assuming that the next `_IO_READ` request will “pick up” where the previous one left off.

bootable

An image can be either bootable or *nonbootable*. A bootable image is one that contains the startup code that the IPL can transfer control to.

bootfile

The part of an OS image that runs the *startup code* and the microkernel.

bound multiprocessing (BMP)

A multiprocessing system where a single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

budget

In *sporadic* scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for `mkifs` specifying the contents and other details of an *image*, or for `mkefs` specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or “cooked” mode. In this mode the character device library performs line-editing operations on each received character. Only when a line is “completely entered” — typically when a carriage return (CR) is received — will the line of data be made available to application processes. Contrast *raw mode*.

channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a *connection* (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using `ChannelCreate()`), and then receives messages from that channel (using `MsgReceive()`). Another thread that wishes to send a message to the first thread must make a connection to that channel by “attaching” to the channel (using `ConnectAttach()`) and then sending data (using `MsgSend()`).

chid

An abbreviation for *channel ID*.

CIFS

Common Internet File System (also known as SMB) — a protocol that allows a client workstation to perform transparent file access over a network to a Windows 95/98/NT server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

CIS

Card Information Structure — a data block that maintains information about flash configuration. The CIS description includes the types of memory devices in the regions, the physical geometry of these devices, and the partitions located on the flash.

coid

An abbreviation for *connection ID*.

combine message

A resource manager message that consists of two or more messages. The messages are constructed as combine messages by the client's C library (e.g. *stat()*, *readblock()*), and then handled as individual messages by the resource manager.

The purpose of combine messages is to conserve network bandwidth and/or to provide support for atomic operations. See also *connect message* and *I/O message*.

connect message

In a resource manager, a message issued by the client to perform an operation based on a pathname (e.g. an *io_open* message). Depending on the type of connect message sent, a context block (see *OCB*) may be associated with the request and will be passed to subsequent I/O messages. See also *combine message* and *I/O message*.

connection

A kernel object used with message passing.

Connections are created by client threads to “connect” to the channels made available by servers. Once connections are established, clients can *MsgSendv()* messages over them. If a number of threads in a process all attach to the same channel, then the one connection is shared among all

the threads. Channels and connections are identified within a process by a small integer.

The key thing to note is that connections and file descriptors (*FD*) are one and the same object. See also *channel* and *FD*.

context

Information retained between invocations of functionality.

When using a resource manager, the client sets up an association or *context* within the resource manager by issuing an *open()* call and getting back a file descriptor. The resource manager is responsible for storing the information required by the context (see *OCB*). When the client issues further file-descriptor based messages, the resource manager uses the OCB to determine the context for interpretation of the client's messages.

cooked mode

See *canonical mode*.

core dump

A file describing the state of a process that terminated abnormally.

critical section

A code passage that *must* be executed “serially” (i.e. by only one thread at a time). The simplest form of critical section enforcement is via a *mutex*.

deadlock

A condition in which one or more threads are unable to continue due to resource contention. A common form of deadlock can occur when one thread sends a message to another, while the other thread sends a message to the first. Both threads are now waiting for each other to reply to the message. Deadlock can be avoided by good design practices or massive kludges — we recommend the good design approach.

device driver

A process that allows the OS and application programs to make use of the underlying hardware in a generic way (e.g. a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for the QNX Neutrino RTOS are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn't affect any other part of the OS — drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won't cause the entire OS to shut down.

discrete (or traditional) multiprocessor system

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

DNS

Domain Name Service — an Internet protocol used to convert ASCII domain names into IP addresses. In QNX Neutrino native networking, `dns` is one of *Qnet's* builtin resolvers.

dynamic bootfile

An OS image built on the fly. Contrast *static bootfile*.

dynamic linking

The process whereby you link your modules in such a way that the Process Manager will link them to the library modules before your program runs. The word “dynamic” here means that the association between your program and the library modules that it uses is done *at load time*, not at linktime. Contrast *static linking*. See also *runtime loading*.

edge-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is “noticed” upon a transition to/from the rising/falling edge of a pulse. Contrast *level-sensitive*.

edited mode

See *canonical mode*.

EOI

End Of Interrupt — a command that the OS sends to the PIC after processing all Interrupt Service Routines (ISR) for that particular interrupt source so that the PIC can reset the processor's In Service Register. See also *PIC* and *ISR*.

EPROM

Erasable Programmable Read-Only Memory — a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g. 12V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. To change a bit from a 0 state into a 1 state can only be accomplished by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period

of time (typically 10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast *flash* and *RAM*.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requestor of the event.

FD

File Descriptor — a client must open a file descriptor to a resource manager via the *open()* function call. The file descriptor then serves as a handle for the client to use in subsequent messages. Note that a file descriptor is the exact same object as a connection ID (*coid*, returned by *ConnectAttach()*).

FIFO

First In First Out — a scheduling policy whereby a thread is able to consume CPU at its priority level without bounds. See also *adaptive*, *round robin*, and *sporadic*.

flash memory

A memory technology similar in characteristics to *EPROM* memory, with the exception that erasing is performed electrically instead of via ultraviolet light, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64 KB at a time) instead of the entire device. Contrast *EPROM* and *RAM*.

FQNN

Fully Qualified Node Name — a unique name that identifies a QNX Neutrino node on a network. The FQNN consists of the nodename plus the node domain tacked together.

garbage collection

Also known as space reclamation, the process whereby a filesystem manager recovers the space occupied by deleted files and directories.

HA

High Availability — in telecommunications and other industries, HA describes a system's ability to remain up and running without interruption for extended periods of time.

handle

A pointer that the resource manager base library binds to the pathname registered via *resmgr_attach()*. This handle is typically used to associate some kind of per-device information. Note that if you use the *iofunc_**() *POSIX layer calls*, you must use a particular *type* of handle — in this case called an *attributes structure*.

hard thread affinity

A user-specified binding of a thread to a set of processors, done by means of a *runmask*. Contrast *soft thread affinity*.

image

In the context of embedded QNX Neutrino systems, an “image” can mean either a structure that contains files (i.e. an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim FFS-2-compatible filesystem (i.e. a flash filesystem image).

inherit mask

A bitmask that specifies which processors a thread's children can run on. Contrast *runmask*.

interrupt

An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.

interrupt handler

See *ISR*.

interrupt latency

The amount of elapsed time between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service routine. Also designated as “ T_{ii} ”. Contrast *scheduling latency*.

interrupt service routine

See *ISR*.

interrupt service thread

A thread that is responsible for performing thread-level servicing of an interrupt.

Since an *ISR* can call only a very limited number of functions, and since the amount of time spent in an *ISR* should be kept to a minimum, generally the bulk of the interrupt servicing work should be done by a thread. The thread attaches the interrupt (via *InterruptAttach()* or *InterruptAttachEvent()*) and then blocks (via *InterruptWait()*), waiting for the *ISR* to tell it to do something (by returning an event of type *SIGEV_INTR*). To aid in minimizing *scheduling latency*, the interrupt service thread should raise its priority appropriately.

I/O message

A message that relies on an existing binding between the client and the resource manager. For example, an *_IO_READ* message depends on the client's having previously established an association (or *context*) with the resource manager by issuing an *open()* and getting back a file descriptor. See also *connect message*, *context*, *combine message*, and *message*.

I/O privileges

A particular right, that, if enabled for a given thread, allows the thread to perform I/O instructions (such as the x86 assembler *in* and *out* instructions). By default, I/O privileges are disabled, because a program with it enabled can wreak havoc on a system. To enable I/O privileges, the thread must be running as *root*, and call *ThreadCtl()*.

IPC

Interprocess Communication — the ability for two processes (or threads) to communicate. The QNX Neutrino RTOS offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), as well as signals.

IPL

Initial Program Loader — the software component that either takes control at the processor's reset vector (e.g. location *0xFFFFFFFF* on the x86), or is a BIOS extension. This component is responsible for setting up the machine into a usable state, such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also *BIOS extension signature* and *startup code*.

IRQ

Interrupt Request — a hardware request line asserted by a peripheral to indicate that it requires servicing by software. The *IRQ* is handled by the *PIC*, which then interrupts the processor, usually causing the processor to execute an *Interrupt Service Routine (ISR)*.

ISR

Interrupt Service Routine — a routine responsible for servicing hardware (e.g. reading and/or writing some device ports), for updating some data structures shared between the ISR and the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.

kernel

See *microkernel*.

level-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast *edge-sensitive*.

linearly mapped

A term indicating that a certain memory component is entirely addressable by the processor. Contrast *bank-switched*.

message

A parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message — the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various “changes of state” that affect when, and for how long, they may run.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mount structure

An optional, well-defined data structure (of type `iofunc_mount_t`) within an `iofunc_*` structure, which contains information used on a per-mountpoint basis (generally used only for filesystem resource managers). See also *attributes structure* and *OCB*.

mountpoint

The location in the pathname space where a resource manager has “registered” itself. For example, the serial port resource manager registers mountpoints for each serial device (`/dev/ser1`, `/dev/ser2`, etc.), and a CD-ROM filesystem may register a single mountpoint of `/cdrom`.

multicore system

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (`pthread_mutex_lock()`) and released (`pthread_mutex_unlock()`) around the code that accesses the shared data (usually a *critical section*). See also *critical section*.

name resolution

In a QNX Neutrino network, the process by which the *Qnet* network manager converts an *FQNN* to a list of destination addresses that the transport layer knows how to get to.

name resolver

Program code that attempts to convert an *FQNN* to a destination address.

nd

An abbreviation for *node descriptor*, a numerical identifier for a node *relative to the current node*. Each node's node descriptor for itself is 0 (`ND_LOCAL_NODE`).

NDP

Node Discovery Protocol — proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.

network directory

A directory in the pathname space that's implemented by the *Qnet* network manager.

NFS

Network FileSystem — a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities

you use for listing and managing files (e.g. `ls`, `cp`, `mv`) operate on the remote files exactly as they do on your local files.

NMI

Nonmaskable Interrupt — an interrupt that can't be masked by the processor. We don't recommend using an NMI!

Node Discovery Protocol

See *NDP*.

node domain

A character string that the *Qnet* network manager tacks onto the nodename to form an *FQNN*.

nodename

A unique name consisting of a character string that identifies a node on a network.

nonbootable

A nonbootable OS image is usually provided for larger embedded systems or for small embedded systems where a separate, configuration-dependent setup may be required. Think of it as a second “filesystem” that has some additional files on it. Since it's nonbootable, it typically won't contain the OS, startup file, etc. Contrast *bootable*.

OCB

Open Control Block (or Open Context Block) — a block of data established by a resource manager during its handling of the client's *open()* function. This context block is bound by the resource manager to this particular request, and is then automatically passed to all subsequent I/O functions generated by the client on the file descriptor returned by the client's *open()*.

package filesystem

A virtual filesystem manager that presents a customized view of a set of files and directories to a client. The “real” files are present on some medium; the package filesystem presents a virtual view of selected files to the client.

partition

A division of CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

pathname prefix

See *mountpoint*.

pathname space mapping

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

persistent

When applied to storage media, the ability for the medium to retain information across a power-cycle. For example, a hard disk is a persistent storage medium, whereas a ramdisk is not, because the data is lost when power is lost.

PIC

Programmable Interrupt Controller — hardware component that handles IRQs. See also *edge-sensitive*, *level-sensitive*, and *ISR*.

PID

Process ID. Also often *pid* (e.g. as an argument in a function call).

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface — the “X” alludes to “UNIX”, on which the interface is based.

POSIX layer calls

Convenient set of library calls for writing resource managers. The POSIX layer calls can handle even more of the common-case messages and functions than the *base layer calls*. These calls are identified by the *iofunc_**() prefix. In order to use these (and we strongly recommend that you do), you must also use the well-defined POSIX-layer *attributes* (*iofunc_attr_t*), *OCB* (*iofunc_ocb_t*), and (optionally) *mount* (*iofunc_mount_t*) structures.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been “preempted” by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY, the lower-priority thread is immediately preempted by the higher-priority thread.

prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent *priority inversion*.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU *on behalf of* the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it's working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one *thread* running in it — this thread is then called the first thread.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.

process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of 1.

pty

Pseudo-TTY — a character-based device that has two “ends”: a master end and a slave end. Data written to the master end shows up on the slave end, and vice versa. These devices are typically used to interface between a program that expects a character device and another program that wishes

to use that device (e.g. the shell and the `telnet` daemon process, used for logging in to a system over the Internet).

pulses

In addition to the synchronous Send/Receive/Reply services, QNX Neutrino also supports fixed-size, nonblocking messages known as pulses. These carry a small payload (four bytes of data plus a single byte code). A pulse is also one form of *event* that can be returned from an ISR or a timer. See *MsgDeliverEvent()* for more information.

Qnet

The native network manager in the QNX Neutrino RTOS.

QoS

Quality of Service — a policy (e.g. `loadbalance`) used to connect nodes in a network in order to ensure highly dependable transmission. QoS is an issue that often arises in high-availability (*HA*) networks as well as realtime control systems.

RAM

Random Access Memory — a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast *flash* and *EPROM*.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data — you don't want any translations of the raw binary stream between the device and the application. Contrast *canonical mode*.

replenishment

In *sporadic* scheduling, the period of time during which a thread is allowed to consume its execution *budget*.

reset vector

The address at which the processor begins executing instructions after the processor's reset line has been activated. On the x86, for example, this is the address `0xFFFFFFFF`.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX Neutrino resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g. serial ports, parallel ports, network cards, disk drives) or virtual (e.g. `/dev/null`, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers*. But unlike device drivers, QNX Neutrino resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also *device driver*.

RMA

Rate Monotonic Analysis — a set of methods used to specify, analyze, and predict the timing behavior of realtime systems.

round robin

A scheduling policy whereby a thread is given a certain period of time to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also *adaptive*, *FIFO*, and *sporadic*.

runmask

A bitmask that indicates which processors a thread can run on. Contrast *inherit mask*.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast *static linking*.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as " T_{sl} ". Contrast *interrupt latency*.

soid

An abbreviation for *server connection ID*.

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

soft thread affinity

The scheme whereby the microkernel tries to dispatch a thread to the processor where it last ran, in an attempt to reduce thread migration from one processor to another, which can affect cache performance. Contrast *hard thread affinity*.

software interrupts

Similar to a hardware interrupt (see *interrupt*), except that the source of the interrupt is software.

sporadic

A scheduling policy whereby a thread's priority can oscillate dynamically between a “foreground” or normal priority and a “background” or low priority. A thread is given an execution *budget* of time to be consumed within a certain *replenishment* period. See also *adaptive*, *FIFO*, and *round robin*.

startup code

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

static bootfile

An image created at one time and then transmitted whenever a node boots. Contrast *dynamic bootfile*.

static linking

The process whereby you combine your modules with the modules from the library to form a single executable that's entirely self-contained. The word “static” implies that it's not going to change — *all* the required modules are already combined into one.

symmetric multiprocessing (SMP)

A multiprocessor system where a single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under the QNX Neutrino RTOS. A thread is a flow of execution; it exists within the context of a *process*.

tid

An abbreviation for *thread ID*.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an *event*, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a *round-robin* or *adaptive* scheduled thread. This period of time is small (on the order of tens of milliseconds); the actual value shouldn't be relied upon by any program (it's considered bad design).

TLB

An abbreviation for *translation look-aside buffer*. To maintain performance, the processor caches frequently used portions of the external memory page tables in the TLB.

TLS

An abbreviation for *thread local storage*.

Index

__BEGIN_DECLS 51
 __BIGENDIAN__ 51
 __builtin_alloca() 147
 __cabsargs, __cabsfargs 242
 __CHAR_SIGNED__ 51
 __CHAR_UNSIGNED__ 51
 __DIOF() 405
 __DION() 405
 __DIOT() 405
 __DIOTF() 405
 __END_DECLS 51
 __FILE__ 161
 __INT_BITS__ 51
 __LINE__ 161
 __LITTLEENDIAN__ 51
 __LONG_BITS__ 51
 __OPTIMIZE__ 51
 __prognose 2183
 __PTR_BITS__ 51
 __QNX__ 51
 __QNXNTO__ 51
 __res_state 2563
 __signalstub() 2977
 __tls() 3405
 _amblksiz 151, 1569, 2702, 3068
 _argc 153
 _argv 154
 _auxv 199
 _btext 237
 _client_able 361
 _client_info 357, 364
 _clockadjust 322
 _clockperiod 329
 _cmdfd() 341
 _cmdname() 342
 _cred_info 358, 365
 _CS_ARCHITECTURE 345
 _CS_DOMAIN 345
 _CS_HOSTNAME 345, 883, 2868
 _CS_HW_PROVIDER 345
 _CS_HW_SERIAL 345
 _CS_LIBPATH 345, 455
 _CS_LOCALE 345
 _CS_MACHINE 345
 _CS_PATH 345
 _CS_RELEASE 345
 _CS_RESOLVE 345
 _CS_SET 346
 _CS_SRPC_DOMAIN 345
 _CS_SYSNAME 346
 _CS_TIMEZONE 346, 3547
 _CS_VERSION 346
 _DCMD_ALL 405
 _DCMD_BLK 405
 _DCMD_CAM 405
 _DCMD_CHR 405
 _DCMD_FSYS 405
 _DCMD_INPUT 405
 _DCMD_IP 405
 _DCMD_MEM 405
 _DCMD_MISC 405
 _DCMD_MIXER 405
 _DCMD_NET 405
 _DCMD_PROC 405
 _DEVCTL_DATA() 1259
 _DEXTRA_FIRST() 2515
 _DEXTRA_NEXT() 2515
 _DEXTRA_VALID() 2515
 _DTYPE_LSTAT 1341, 1384, 2515
 _DTYPE_NONE 2515
 _DTYPE_STAT 2515
 _edata 491
 _end 494
 _etext 549
 _exit() 596
 _fdinfo 1266, 1269
 _FDINFO_FLAG_LOCALPATH 1210, 1269
 _FP_EXC_DENORMAL 690, 693
 _FP_EXC_DIVZERO 690, 693
 _FP_EXC_INEXACT 690, 693
 _FP_EXC_INVALID 690, 693
 _FP_EXC_OVERFLOW 690, 693
 _FP_EXC_UNDERFLOW 690, 693
 _FP_PREC_DOUBLE 695
 _FP_PREC_DOUBLE_EXTENDED 695
 _FP_PREC_EXTENDED 695
 _FP_PREC_FLOAT 695
 _FP_ROUND_NEAREST 697
 _FP_ROUND_NEGATIVE 697
 _FP_ROUND_POSITIVE 697
 _FP_ROUND_ZERO 697
 _FTYPE_ANY 1190, 1195, 1196, 2580
 _FTYPE_LINK 1190, 1195, 1196, 2580
 _FTYPE_MOUNT 1190, 1195, 1196, 2580
 _FTYPE_MQUEUE 1190, 1195, 1196, 2580
 _FTYPE_PIPE 1190, 1195, 1196, 2580
 _FTYPE_SEM 1190, 1195, 1196, 2580
 _FTYPE_SHMEM 1190, 1195, 1196, 2580
 _FTYPE_SOCKET 1190, 1195, 1196, 2580
 _FTYPE_SYMLINK 1190, 1195, 1196, 2581
 _input_line_max 1144
 _intr_v86() 1185
 _intrspin_t 1162, 1172, 1174, 1179
 _IO_ACL_CHK 1213
 _IO_ACL_GET 1213
 _IO_ACL_SET 1213
 _IO_CHMOD 1234, 1237
 _IO_CHOWN 1239, 1242
 _IO_CLOSE 1249, 1251
 _IO_COMBINE_FLAG 1213, 1235, 1240, 1250, 1259, 1269, 1287, 1295, 1303, 1328, 1333, 1340, 1351, 1356, 1363, 1376, 1383

- _io_connect 1189
- _IO_CONNECT 1189, 1325
- _IO_CONNECT_COMBINE 1189
- _IO_CONNECT_COMBINE_CLOSE 1189
- _IO_CONNECT_EFLAG_DIR 1193, 1196
- _IO_CONNECT_EFLAG_DOT 1193, 1197
- _IO_CONNECT_EXTRA_LINK 1193
- _IO_CONNECT_EXTRA_MOUNT 1194
- _IO_CONNECT_EXTRA_MOUNT_OCB 1194
- _IO_CONNECT_EXTRA_MQUEUE 1193
- _IO_CONNECT_EXTRA_NONE 1193
- _IO_CONNECT_EXTRA_PROC_SYMLINK 1194
- _IO_CONNECT_EXTRA_RENAME 1194
- _IO_CONNECT_EXTRA_RESMGR_LINK 1193
- _IO_CONNECT_EXTRA_SEM 1193
- _IO_CONNECT_EXTRA_SOCKET 1193
- _IO_CONNECT_EXTRA_SYMLINK 1193
- _io_connect_ftype_reply 1195
- _IO_CONNECT_LINK 1189
- _io_connect_link_reply 1196
- _IO_CONNECT_MKNOD 1189
- _IO_CONNECT_MOUNT 1190
- _IO_CONNECT_OPEN 1189
- _IO_CONNECT_READLINK 1189
- _IO_CONNECT_RENAME 1189, 1347
- _IO_CONNECT_RET_LINK 1322
- _IO_CONNECT_RSVD_UNBLOCK 1190
- _IO_CONNECT_UNLINK 1189
- _IO_DEVCTL 1257, 1261, 1264
- _IO_DEVCTL_VERIFY_ACC_IEXEC 1263
- _IO_DEVCTL_VERIFY_ACC_IREAD 1263
- _IO_DEVCTL_VERIFY_ACC_ISGID 1263
- _IO_DEVCTL_VERIFY_ACC_ISUID 1263
- _IO_DEVCTL_VERIFY_ACC_IWRITE 1264
- _IO_DEVCTL_VERIFY_DIR 1264
- _IO_DEVCTL_VERIFY_LEN 1264
- _IO_DEVCTL_VERIFY_OCB_READ 1264
- _IO_DEVCTL_VERIFY_OCB_WRITE 1264
- _IO_DEVCTL_VERIFY_PRIV 1264
- _IO_FDINFO 1267, 1268
- _IO_FLAG_RD 1190, 1320, 1328, 1358
- _IO_FLAG_WR 1190, 1320, 1328, 1358
- _IO_LOCK 1281
- _IO_LSEEK 1286, 1289
- _IO_MMAP 1294, 1298
- _IO_OPENFD 1327, 1330
- _IO_OPENFD_NONE 1329
- _IO_OPENFD_PIPE 1329
- _IO_OPENFD_RESERVED 1329
- _IO_PATHCONF 1332, 1335
- _IO_READ 1337, 1340
- _IO_SET_CONNECT_RET() 1322
- _IO_SET_PATHCONF_VALUE() 1333
- _IO_SET_READ_NBYTES() 1340
- _IO_SET_WRITE_NBYTES() 1383
- _IO_SPACE 1351
- _IO_STAT 1353, 1355
- _IO_SYNC 1360, 1362
- _IO_UTIME 1376, 1378
- _IO_WRITE 1380
- _IO_XFLAG_BLOCK 1341, 1384
- _IO_XFLAG_DIR_EXTRA_HINT 1341, 1384
- _IO_XFLAG_NONBLOCK 1341, 1384
- _IO_XTYPE_MQUEUE 1341, 1384
- _IO_XTYPE_NONE 1341, 1384
- _IO_XTYPE_OFFSET 1341, 1384
- _IO_XTYPE_READCOND 1341, 1384
- _IO_XTYPE_REGISTRY 1341, 1384
- _IO_XTYPE_TCPIP 1341, 1384
- _IO_XTYPE_TCPIP_MSG 1341, 1384
- _IOFBF 2920
- _IOLBF 2920
- _IONBF 2920
- _itimer 3461
- _MAJOR_BLK_PREFIX 2689
- _MAJOR_CHAR_PREFIX 2689
- _MAJOR_DEV 2689
- _MAJOR_FSYS 2689
- _MAJOR_PATHMGR 2689
- _MFLAG_OCB 845, 1691, 1694
- _MOUNT_AFTER 1691, 1694
- _MOUNT_BEFORE 1691, 1694
- _MOUNT_ENUMERATE 1692, 1694
- _MOUNT_FORCE 1691, 1694, 3559
- _MOUNT IMPLIED 845, 1694
- _MOUNT_NOATIME 784, 1691, 1694, 3171
- _MOUNT_NOCREAT 784, 1691, 1694, 3171
- _MOUNT_NOEXEC 784, 1258, 1691, 1694, 3171
- _MOUNT_NOSUID 784, 1258, 1691, 1694, 3171
- _MOUNT_OFF32 784, 1691, 1694, 3171
- _MOUNT_OPAQUE 1691, 1694
- _MOUNT_READONLY 784, 1258, 1691, 1694, 3171
- _MOUNT_REMOUNT 1691, 1694
- _MOUNT_UNMOUNT 1691, 1694
- _msg_info 1731, 1745
- _NOTIFY_ACTION_POLL 1388, 1389
- _NOTIFY_ACTION_POLLARM 1389
- _NOTIFY_ACTION_TRANARM 1389
- _NOTIFY_COND_EXTEN 1300
- _NOTIFY_COND_INPUT 1300, 1303, 1387
- _NOTIFY_COND_MASK 1388
- _NOTIFY_COND_OBAND 1300, 1303, 1387
- _NOTIFY_COND_OUTPUT 1300, 1303, 1387
- _NOTIFY_CONDE_ERR 1301, 1387
- _NOTIFY_CONDE_HUP 1301, 1387
- _NOTIFY_CONDE_NVAL 1301, 1387
- _NOTIFY_CONDE_PRI 1301, 1387
- _NOTIFY_CONDE_RDBAND 1301, 1387
- _NOTIFY_CONDE_RDNORM 1301, 1387
- _NOTIFY_CONDE_WRBAND 1301, 1387
- _NOTIFY_CONDE_WRNORM 1301, 1387
- _NOTIFY_DATA_MASK 1388
- _NTO_CHF_COID_DISCONNECT 280, 1843
- _NTO_CHF_DISCONNECT 281, 1843
- _NTO_CHF_FIXED_PRIORITY 279, 281, 1734
- _NTO_CHF_NET_MSG 281
- _NTO_CHF_PRIVATE 281, 354, 2146
- _NTO_CHF_REPLY_LEN 281, 1732
- _NTO_CHF_SENDER_LEN 281, 1732
- _NTO_CHF_THREAD_DEATH 281
- _NTO_CHF_UNBLOCK 282, 1732, 1843
- _NTO_CI_BKGND_PGRP 357, 364
- _NTO_CI_ENDIAN_BIG 357, 364
- _NTO_CI_FULL_GROUPS 358, 364

_NTO_CI_ORPHAN_PGRP 357, 364
_NTO_CI_STOPPED 357, 364
_NTO_CI_UNABLE 357, 364
_NTO_CLIENTINFO_GETGROUPS 360, 363, 1244, 1246
_NTO_COF_ASYNC 374
_NTO_COF_CLOEXEC 352, 369, 373
_NTO_COF_DEAD 373
_NTO_COF_GLOBAL 374
_NTO_COF_INSECURE 369, 1738
_NTO_COF_NONBLOCK 374
_NTO_COF_NOSHARE 373
_NTO_IC_LATENCY 1159
_NTO_INTR_CLASS_EXTERNAL 1148
_NTO_INTR_CLASS_SYNTHETIC 1148
_NTO_INTR_FLAGS_END 1151, 1156, 1169, 1172
_NTO_INTR_FLAGS_PROCESS 1151, 1152, 1156, 1169
_NTO_INTR_FLAGS_TRK_MSK 1151, 1169
_NTO_INTR_SPARE 1148
_NTO_KEYDATA_CALCULATE 1747, 1749
_NTO_KEYDATA_VERIFY 1747, 1750
_NTO_MI_CONSTRAINED 1732
_NTO_MI_ENDIAN_BIG 1732
_NTO_MI_ENDIAN_DIFF 1732
_NTO_MI_NET_CRED_DIRTY 1732
_NTO_MI_UNBLOCK_REQ 1732
_NTO_RESET_OVERRUNS 3456
_NTO_SCTL_GETPRIOCEILING 3286
_NTO_SCTL_SETEVENT 3286
_NTO_SCTL_SETPRIOCEILING 3286
_NTO_SIDE_CHANNEL 352
_NTO_SYNC_COND 3305
_NTO_SYNC_MUTEX_FREE 3305
_NTO_SYNC_SEM 3305
_NTO_TCTL_ALIGN_FAULT 3409
_NTO_TCTL_IO 247, 254, 1148, 1155, 1160, 1162, 1164, 1166, 1172, 1174, 1176, 1179, 1181, 2144, 2979, 3409, 3518, 3519
_NTO_TCTL_IO_PRIV 2144, 3410
_NTO_TCTL_NAME 3410
_NTO_TCTL_ONE_THREAD_CONT 3412
_NTO_TCTL_ONE_THREAD_HOLD 3412
_NTO_TCTL_RCM_GET_AND_SET 3412
_NTO_TCTL_RUNMASK 3412
_NTO_TCTL_RUNMASK_GET_AND_SET 3413
_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT 3413
_NTO_TCTL_THREADS_CONT 3415
_NTO_TCTL_THREADS_HOLD 3415
_NTO_THREAD_NAME_MAX 2398
_NTO_TI_ABSOLUTE 3457
_NTO_TI_ACTIVE 3457
_NTO_TI_EXPIRED 3457
_NTO_TI_PRECISE 3457
_NTO_TI_REPORT_TOLERANCE 3456
_NTO_TIMEOUT_CONDVAR 3466
_NTO_TIMEOUT_INTR 3466
_NTO_TIMEOUT_JOIN 3466
_NTO_TIMEOUT_MUTEX 3466
_NTO_TIMEOUT_RECEIVE 3466
_NTO_TIMEOUT_REPLY 3466
_NTO_TIMEOUT_SEM 3466
_NTO_TIMEOUT_SEND 3466
_NTO_TIMEOUT_SIGSUSPEND 3466
_NTO_TIMEOUT_SIGWAITINFO 3467
_NTO_TIMER_SEARCH 3456
_NTO_TRACE_ADDALLCLASSES 3516
_NTO_TRACE_ADDCLASS 3516
_NTO_TRACE_ADDCLASSEVHANDLER 3519
_NTO_TRACE_ADDEVENT 3516
_NTO_TRACE_ADDEVENTHANDLER 3518
_NTO_TRACE_ALLOCBUFFER 3513
_NTO_TRACE_CLRCLASSPID 3516
_NTO_TRACE_CLRCLASSTID 3516
_NTO_TRACE_CLREVENTPID 3516
_NTO_TRACE_CLREVENTTID 3516
_NTO_TRACE_DEALLOCBUFFER 3513
_NTO_TRACE_DELALLCLASSES 3516
_NTO_TRACE_DELCLASS 3516
_NTO_TRACE_DELCLASSEVHANDLER 3519
_NTO_TRACE_DELEVENT 3516
_NTO_TRACE_DELEVENTHANDLER 3518
_NTO_TRACE_FIPID 3517
_NTO_TRACE_FITID 3517
_NTO_TRACE_FLUSHBUFFER 3514
_NTO_TRACE_FMPID 3517
_NTO_TRACE_FMTID 3517
_NTO_TRACE_GETCPU() 3518
_NTO_TRACE_GETEVENT_C() 3518
_NTO_TRACE_GETEVENT() 3518
_NTO_TRACE_INSERTCUSEREVENT 3520
_NTO_TRACE_INSERTEVENT 3519
_NTO_TRACE_INSERTUSEREVENT 3520
_NTO_TRACE_INSERTUSRSTREVENT 3520
_NTO_TRACE_QUERYEVENTS 3514
_NTO_TRACE_SETALLCLASSESFAST 3515
_NTO_TRACE_SETALLCLASSESWIDE 3515
_NTO_TRACE_SETBUFFER 3514
_NTO_TRACE_SETCLASSFAST 3515
_NTO_TRACE_SETCLASSPID 3516
_NTO_TRACE_SETCLASSTID 3516
_NTO_TRACE_SETCLASSWIDE 3515
_NTO_TRACE_SETEVENT_C() 3518
_NTO_TRACE_SETEVENT() 3518
_NTO_TRACE_SETEVENTFAST 3515
_NTO_TRACE_SETEVENTPID 3516
_NTO_TRACE_SETEVENTTID 3516
_NTO_TRACE_SETEVENTWIDE 3515
_NTO_TRACE_SETLINEARMODE 3514
_NTO_TRACE_SETRINGMODE 3514
_NTO_TRACE_START 3514
_NTO_TRACE_STARTNOSTATE 3514
_NTO_TRACE_STOP 3514
_NTO_TRACE_SYS_ADDRESS 3496
_NTO_TRACE_SYS_FUNC_EXIT 3492, 3494
_NTO_TRACE_SYSTEM 3492, 3494, 3496
_NTO_TRACE_USER 3498, 3502, 3504, 3506, 3508
_NTO_TRACE_USERFIRST 3498, 3502, 3504, 3506, 3508
_NTO_TRACE_USERLAST 3498, 3502, 3504, 3506, 3508
_NTO_VERSION 51
_PATH_UTMP 527
_PC_ASYNC_IO 699, 1936
_PC_CHOWN_RESTRICTED 699, 1936
_PC_LINK_DIR 700, 1937
_PC_LINK_MAX 700, 1937
_PC_MAX_CANON 700, 1937

_PC_MAX_INPUT 700, 1937
_PC_NAME_MAX 700, 1937
_PC_NO_TRUNC 700, 1937
_PC_PATH_MAX 700, 1937
_PC_PIPE_BUF 700, 1937
_PC_PRIO_IO 700, 1937
_PC_SYNC_IO 700, 1937
_PC_VDISABLE 700, 1937, 3384
_PCCARD_ARM_INSERT_REMOVE 1949
_PCCARD_DEV_AIMS 1949
_PCCARD_DEV_ALL 1949
_PCCARD_DEV_FIXED_DISK 1949
_PCCARD_DEV_GPIB 1949
_PCCARD_DEV_MEMORY 1949
_PCCARD_DEV_NETWORK 1949
_PCCARD_DEV_PARALLEL 1949
_PCCARD_DEV_SCSI 1949
_PCCARD_DEV_SERIAL 1949
_PCCARD_DEV_SOUND 1949
_PCCARD_DEV_VIDEO 1949
_pccard_info 1956
_PCCARD_MEMTYPE_ATTRIBUTE 1960
_PCCARD_MEMTYPE_COMMON 1960
_POSIX_AIO_MAX 1519
_POSIX_CHOWN_RESTRICTED 293, 615, 1503
_POSIX_LOGIN_NAME_MAX 892
_POSIX_THREAD_SAFE_FUNCTIONS 855, 892
_pulse 2440
_PULSE_CODE_COIDDEATH 280, 2440
_PULSE_CODE_DISCONNECT 281, 2440
_PULSE_CODE_MAXAVAIL 1789, 2440, 2962
_PULSE_CODE_MINAVAIL 1789, 2440, 2962
_PULSE_CODE_NET_ACK 2440
_PULSE_CODE_NET_DETACH 2440
_PULSE_CODE_NET_UNBLOCK 2440
_PULSE_CODE_THREADDEATH 281, 441, 2440
_PULSE_CODE_UNBLOCK 282, 2440
_PULSE_SUBTYPE 2440
_PULSE_TYPE 2440
_RESMGR_CONNECT_NFUNCS 1271
_RESMGR_DEFAULT 1366
_RESMGR_DETACH_ALL 2600
_RESMGR_DETACH_CLOSE 2600
_RESMGR_DETACH_PATHNAME 2600
_RESMGR_FLAG_AFTER 1691, 1694, 2584
_RESMGR_FLAG_BEFORE 1691, 1694, 2584
_RESMGR_FLAG_DIR 2585, 2640
_RESMGR_FLAG_FTYPEALL 2585
_RESMGR_FLAG_FTYPEONLY 2586
_RESMGR_FLAG_OPAQUE 1691, 1694, 2586
_RESMGR_FLAG_SELF 2586
_RESMGR_IO_NFUNCS 1271
_RESMGR_NOREPLY 1366
_RESMGR_NPARTS() 2632
_RESMGR_PATHNAME_LOCALPATH 2639
_RESMGR_PTR() 2641
_RESMGR_STATUS() 2643
_SC_AIO_PRIO_DELTA_MAX 3308
_SC_ARG_MAX 3308
_SC_CHILD_MAX 3308
_SC_CLK_TCK 3309
_SC_DELAYTIMER_MAX 3309
_SC_GETGR_R_SIZE_MAX 856, 3309
_SC_GETPW_R_SIZE_MAX 932, 936, 940, 971, 974, 3309
_SC_JOB_CONTROL 3309
_SC_NGROUPS_MAX 864, 2864, 3309
_SC_OPEN_MAX 3309
_SC_PAGESIZE 908, 2217, 2233, 3309
_SC_RCT_MEM 3309
_SC_RCT_SCOID 3309
_SC_SAVED_IDS 3309
_SC_SEM_NSEMS_MAX 3310
_SC_SIGQUEUE_MAX 3310
_SC_THREAD_STACK_MIN 3310
_SC_TZNAME_MAX 3310
_SC_VERSION 3310
_scalloc() 2709
_sched_info 2778
_server_info 373
_sfree() 2923
_SIGMAX 2981
_SIGMIN 2981
_sleepon_broadcast() 3030
_sleepon_destroy() 3032
_sleepon_init() 3034
_sleepon_lock() 3036
_sleepon_signal() 3038
_sleepon_unlock() 3040
_sleepon_wait() 3042
_SLOG_CRITICAL 3064
_SLOG_DEBUG1 3064
_SLOG_DEBUG2 3064
_SLOG_ERROR 3064
_SLOG_INFO 3064
_SLOG_NOTICE 3064
_SLOG_SETCODE() 3063, 3630
_SLOG_SHUTDOWN 3064
_SLOG_WARNING 3064
_smalloc() 3068
_srealloc() 3160
_SYS_GETPAGESIZES_HIGHUSAGE 909
_syspage_ptr 3329
_thread_attr 3403
_thread_local_storage 3405
_thread_name 3410
_timer_info 3457
_TRACE_GET_BUFFNUM() 1172
_TRACE_GET_BUFFSEQ() 1172
_TRACEPARSER_DEBUG_* 3531
_TRACEPARSER_INFO_* 3534
-fno-builtin 147
.rhosts 1420, 2696
/dev/name/global 1843
/dev/name/local 1843
/dev/zero 1659
/etc/autoconnect 1869
/etc/hosts 497, 866, 867, 870, 871, 872, 873, 877, 878, 882, 1019
/etc/hosts.equiv 1420, 2696
/etc/networks 898, 900, 902, 1876, 2884
/etc/protocols 523, 924, 926, 928, 2184, 2892
/etc/resolv.conf 2565, 2567, 2570, 2573, 2576, 2579

/etc/services 525, 952, 954, 956, 2842, 2905

64-bit addressing, processes and 2043, 3094

8-bit characters, reading 1677

8086 mode, virtual 1185

A

abort() 54

abs() 56

absolute values 56, 242, 606, 1500, 1525

 complex number 242

 floating point 606

 integer 56

 long integer 1500

 long long integer 1525

accept() 58, 3359, 3766

access control lists (ACLs) 64, 66, 68, 70, 72, 74, 76, 78,
 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102,
 104, 106, 108, 110, 112, 114, 116, 1212, 1215

 copying 72, 74

 creating from text 86

 duplicating 82

 entries 70, 76, 78, 88

 copying 70

 creating 76

 deleting 78

 getting 88

 freeing 84

 getting 90, 92

 group class mask 66

 calculating 66

 initializing 100

 permission set 64

 adding 64

 permissions 68, 80, 94, 106

 clearing 68

 deleting 80

 getting 94

 setting 106

 qualifiers 96, 108

 getting 96

 setting 108

 setting 102, 104

 size of 112

 tag type 98, 110

 getting 98

 setting 110

 text form 114

 validating 116, 1212, 1215

access() 61

ACCOUNTING 3595

acl_add_perm() 64

acl_calc_mask() 66

acl_clear_perms() 68

acl_copy_entry() 70

acl_copy_ext() 72

acl_copy_int() 74

acl_create_entry() 76

acl_delete_entry() 78

acl_delete_perm() 80

acl_dup() 82

ACL_EXECUTE 64, 80

ACL_FIRST_ENTRY 88

acl_free() 84

acl_from_text() 86

acl_get_entry() 88

acl_get_fd() 90

acl_get_file() 92

acl_get_permset() 94

acl_get_qualifier() 96

acl_get_tag_type() 98

ACL_GROUP 98, 110, 116

ACL_GROUP_OBJ 98, 110, 116

acl_init() 100

ACL_MASK 98, 110, 116

ACL_NEXT_ENTRY 88

ACL_OTHER 98, 110, 116

ACL_READ 64, 80

acl_set_fd() 102

acl_set_file() 104

acl_set_permset() 106

acl_set_qualifier() 108

acl_set_tag_type() 110

acl_size() 112

acl_to_text() 114

ACL_TYPE_ACCESS 92, 104

ACL_USER 98, 110, 116

ACL_USER_OBJ 98, 110, 116

acl_valid() 116

ACL_WRITE 64, 80

acos(), acosf(), acosl() 118

acosh(), acoshf(), acoshl() 120

ACTION 1021

adaptive partitions 281, 330, 2041, 2043, 2059, 2061,
 2072, 2745, 2746, 2747, 2748, 2749, 2750,
 2751, 2752, 2753, 2755, 2756, 2757, 2759,
 2762, 2763, 2764, 2767, 2768, 2769, 2770,
 2771, 2965

 averaging window size, setting 330, 2747

 bankruptcy, handling 2748

 budgets 2749, 2769, 2770

 zero, preventing 2769, 2770

 zero, setting on bankruptcy 2749

 CPU time, getting 2759

 creating 2751

 IDs, finding for a partition 2755

 inheritance, preventing 281

 locking 2768

 maximum number of 2753

 names 2752

 assigning 2752

 maximum length 2752

 parameters, setting 2757

 processes 2041, 2043, 2059, 2061, 2072, 2756, 2771

 getting 2072, 2771

 setting 2041, 2043, 2059, 2061, 2756

 querying 2753

 scheduler 2745, 2746, 2750

 controlling 2745

 parameters, getting 2746

 parameters, setting 2750

 scheduling policies 2748

 security, setting 2767, 2768

- adaptive partitions (*continued*)
 - System 2751
 - threads 2756, 2762, 2763, 2764, 2770, 2965
 - critical 2762, 2763, 2965
 - determining 2764
 - joining 2756, 2770
- address space layout randomization (ASLR) 1570
- Address Space Layout Randomization (ASLR) 2041, 2043
- addresses 122, 719, 814, 818, 894, 1073, 1075, 1077, 1078, 1089, 1091, 1094, 1096, 1100, 1101
 - hosts 1094, 1096
 - strings, converting to/from 1094, 1096
 - IP 1073, 1075, 1089, 1091, 1094, 1096
 - strings, converting to/from 1073, 1075, 1089, 1091, 1094, 1096
 - IPv6 1100
 - link-local 1101
 - local network 1077, 1078
 - IP addresses, converting to/from 1078
 - IP addresses, extracting from 1077
 - nodenames, translating 894
 - scoped 1101
 - site-local 1101
 - sockets 122, 719, 814, 818
- addrinfo 122, 719, 814, 818
 - errors 814
 - freeing 719
 - getting 818
- advisory locks, *See* files, locking
- AF_INET 209, 866, 869, 872, 898, 1019, 1045, 1093, 1096, 1391, 1876, 3359, 3553
 - must set af_family before calling bind() 209
- AF_INET6 872, 1047, 1093, 1096, 1412
- AF_LOCAL 3577
- AF_UNSPEC 2492, 2664
- AH (Authentication Header) 1396
- AIMS (Auto Incrementing Mass Storage) 1949
- AIO_ALLDONE 125
- aio_cancel() 124
- AIO_CANCELED 125
- aio_error() 127
- aio_fsync() 129
- AIO_NOTCANCELED 125
- aio_read(), aio_read64() 131
- aio_return() 134
- aio_suspend() 136
- aio_write(), aio_write64() 138
- aiocb 1520
- aiocb, aiocb64 141
- alarm() 144
- alarms, scheduling 144, 3550
- aligned memory, allocating 1603, 2033
- alignment, setting for a process 2041, 2043, 3094
- alloca() 147
- alphabetic, testing a character for 1423, 1457
- alphanumeric, testing a character for 1421, 1455
- alphasort() 149
- anonymous memory, allocating 1659
- ANSI classification 46
- application groups 2043, 2986, 3094
 - creating new IDs 2043, 3094
 - signalling 2986
- APS_INIT_DATA() 2746
- APS_PARTITION_NAME_LENGTH 2752
- APS_QCRIT_BILL_AS_CRITICAL 2765
- APS_QCRIT_PERM_CRITICAL 2765
- APS_QCRIT_RUNNING_CRITICAL 2765
- APS_SYSTEM_PARTITION_NAME 2751
- arccosines 118
- architecture, instruction set 345
- arcsines 157
- arctangents 163, 165
- argument lists, variable-length 3602, 3605, 3607, 3608
- arguments to main() 153, 154, 199, 903
 - parsing 903
- ARM_CPU_FLAG_NEON 3326
- arrays 216, 255, 1510, 1552, 2476, 2709
 - allocating 255, 2709
 - quick-sorting 2476
 - searching 216, 1510, 1552
- ASCII, testing a character for 1425
- asctime(), asctime_r() 155
- asin(), asinf(), asinl() 157
- asinh(), asinhf(), asinhl() 159
- ASLR (address space layout randomization) 1570
- ASLR (Address Space Layout Randomization) 2041, 2043
- assert() 161
- asynchronous I/O 124, 127, 129, 131, 134, 136, 138, 141, 699, 1936, 3308
 - canceling 124
 - control block 141
 - error status, getting 127
 - file, synchronizing 129
 - priority for (_SC_AIO_PRIO_DELTA_MAX) 3308
 - reading 131
 - return status, getting 134
 - support for (_PC_ASYNC_IO) 699, 1936
 - waiting for completion 136
 - writing 138
- atan(), atanh(), atanl() 163
- atan2(), atan2f() 165
- atanh(), atanhf(), atanhhl() 167
- atexit() 169
- atof() 171
- atoh() 173
- atoi() 175
- atol(), atoll() 177
- atomic operations 179, 181, 183, 185, 187, 189, 191, 193, 195, 197
 - addition 179, 181
 - bits 183, 185, 187, 189, 195, 197
 - clearing 183, 185
 - setting 187, 189
 - togglng 195, 197
 - subtraction 191, 193
- atomic_add_value() 181
- atomic_add() 179
- atomic_clr_value() 185
- atomic_clr() 183
- atomic_set_value() 189
- atomic_set() 187
- atomic_sub_value() 193
- atomic_sub() 191
- atomic_toggle_value() 197

atomic_toggle() 195
 Authentication Header (AH) 1396
 Auto Incrementing Mass Storage (AIMS) 1949

B

background processes 394, 2156, 2159, 2164
 termination, notification of 2159, 2164
 backtraces 219, 221, 223, 225, 227, 229, 231, 234, 236
 accessor 225
 releasing 225
 addresses, formatting 231
 addresses, translating 234
 collecting 219
 flags, setting 227
 initializing 221
 memory map 223, 236
 loading 223
 unloading 236
 memory maps 229
 formatting 229
 bankruptcies 2748
 handling 2748
 barriers 2237, 2239, 2243, 2245, 2247, 2249
 attributes 2239, 2243, 2245, 2247, 2249
 destroying 2243
 initializing 2247
 process-shared 2245, 2249
 destroying 2237
 initializing 2239
 basename() 202
 bcmp() 204
 bcopy() 206
 Berkeley Packet Filter (BPF) 1206
 Bessel functions 1486, 1488, 1490, 3754, 3756, 3758
 first kind 1486, 1488, 1490
 second kind 3754, 3756, 3758
 big endian 51, 498, 500, 502, 510, 512, 514, 516, 518,
 520, 1063, 1067, 1626, 1927, 1931, 3270,
 3561, 3563, 3565, 3567, 3568, 3569
 __BIGENDIAN__ manifest 51
 little endian, converting to/from 510, 512, 514, 516,
 518, 520, 3270
 messages 1626
 native format, converting to/from 498, 500, 502
 ports 1063, 1067, 1927, 1931
 reading from 1063, 1067
 writing to 1927, 1931
 unaligned values 3561, 3563, 3565, 3567, 3568, 3569
 accessing safely 3567, 3568, 3569
 writing safely 3561, 3563, 3565
 binary search 216
 bind() 208, 3359, 3766
 bindresvport() 211
 BIOCGDLTLIST 1206
 BIOCSETF 1206
 BIOS (PCI), determining if present 1988
 bits 183, 185, 187, 189, 195, 197, 644
 atomic operations 183, 185, 187, 189, 195, 197
 clearing 183, 185
 setting 187, 189
 toggling 195, 197

bits (*continued*)
 set, finding first 644
 blank, testing a character for 1429, 1459
 block buffering, setting for stream I/O 2847
 block special devices 2507, 3743
 reading 2507
 writing 3743
 blocks 2506, 3061, 3068, 3742
 allocating 3068
 reading from a file 2506
 system message log, writing to 3061
 writing to a file 3742
 blocksize, filesystem 784, 3170
 BOOT_TIME 987, 3594
 booting 3319, 3327
 time since 3327
 BPF (Berkeley Packet Filter) 1206
 break condition, asserting 3362
 break pointer 151, 2702
 advancing 2702
 increment 151
 brk() 213
 BRKINT 3379
 bsearch() 216
 BSS data 491, 494
 bt_accessor_t 221
 bt_get_backtrace() 219
 bt_init_accessor() 221
 bt_load_memmap() 223
 BT_PROCESS 219
 bt_release_accessor() 225
 BT_SELF 219, 221
 bt_set_flags() 227
 bt_sprn_memmap() 229
 bt_sprnf_addrs() 231
 BT_THREAD 219, 221
 bt_translate_addrs() 234
 bt_unload_memmap() 236
 BTF_LIVE_BACKTRACE 227
 btowc() 238
 budgets 2749, 2757, 2769, 2770
 setting 2757
 zero 2749, 2769, 2770
 bankruptcy, setting on 2749
 preventing 2769, 2770
 buffers 700, 1937, 2845, 2847, 2878, 2921, 3345, 3513
 canonical input 700, 1937
 raw input 700, 1937
 stream I/O 2845, 2847, 2878, 2921, 3345
 block 2847
 flushing 3345
 line 2878
 trace events 3513
 BUFSIZ 2845
 bus mastering, enabling 1970
 busy-waiting 1857, 1859, 1861, 1863, 1865
 bytes 204, 206, 239, 1026, 1027, 1583, 1585, 1605,
 1611, 1618, 1620, 1622, 1896, 1897, 2131,
 2470, 2502, 3737
 comparing 204
 copying 206, 1605, 1611, 1618
 overlapping objects 1618

bytes (*continued*)
 multibyte character, number of bytes in 1583, 1585
 reading 2131, 2502
 reordering 1026, 1027, 1896, 1897
 setting 1620, 1622
 writing 2470, 3737
 zeroing 239
 bzero() 239

C

C_ANY 2566, 2569, 2572, 2575
 C_CHAOS 2566, 2569, 2572, 2575
 C_HS 2566, 2569, 2572, 2575
 C_IN 2566, 2569, 2572, 2575
 C++ programs 51
 end of C code 51
 start of C code 51
 cabs(), cabsf() 242
 cache coherency 244, 246, 248, 253
 freeing resources 244
 initializing 248
 lines 246, 253
 flushing 246
 invalidating 253
 cache_fini() 244
 CACHE_FLUSH() 246
 CACHE_INIT_FLAG_IGNORE_SCAN 248
 cache_init() 248
 CACHE_INVALID() 253
 CACHE_MSYNCR 247, 254
 CACHE_NOP 247, 254
 CACHE_OP_RATE_CALLOUT 249
 CACHE_OP_RATE_INLINE 249
 CACHE_OP_RATE_MSYNCR 249
 CACHE_OP_RATE_SNOOP 249
 calendar times 1531, 1533, 1649, 3425, 3476
 current 3425
 local times, converting to/from 1531, 1533, 1649
 tm 3476
 calloc() 255
 cancellation 2253, 2255, 2392, 2394, 2432, 3400, 3404
 cleanup handlers 2253, 2255
 points 2432, 3404
 creating 2432
 state 2392, 3400
 type 2394, 3400
 canonical input buffer 700, 1937
 Card Information Structure (CIS), reading 1961
 CardBus cards 1997
 cbrt(), cbrtf() 258
 ceil(), ceilf() 260
 cfgetispeed() 262
 cfgetospeed() 264
 CFGFILE_APPEND 267
 CFGFILE_CREAT 267
 CFGFILE_EXCL 267
 CFGFILE_RDONLY 267
 CFGFILE_RDWR 267
 CFGFILE_TRUNC 268
 CFGFILE_WRONLY 267
 cfopen() 267

cfmakeraw() 270
 cfree() 272
 cfsetispeed() 274
 cfsetospeed() 276
 ChannelCreate() 2146
 ChannelCreate(), ChannelCreate_r() 278
 ChannelDestroy(), ChannelDestroy_r() 284
 channels 278, 279, 280, 281, 284, 351, 1736, 1760,
 1766, 1769, 1771, 1776, 1779, 1782, 1786,
 1790, 1794, 1798, 1802, 1806, 1810, 1814,
 1842, 1850
 creating 278, 1842
 destroying 284, 1850
 events, delivering 1736
 flags 280
 ID 279
 messages 1760, 1771, 1776, 1779, 1782, 1786,
 1794, 1798, 1802, 1806, 1810, 1814
 receiving 1760, 1771
 replying 1776, 1779
 sending 1782, 1786, 1794, 1798, 1802, 1806,
 1810, 1814
 private 281
 pulses 1766, 1769, 1790
 receiving 1766, 1769
 sending 1790
 side channels 351
 CHAR_MAX 1528
 character device terminal drivers, providing session support to
 2176
 characters 41, 51, 238, 645, 648, 652, 700, 704, 707,
 823, 825, 827, 829, 1421, 1423, 1425, 1429,
 1431, 1433, 1439, 1443, 1447, 1449, 1451,
 1453, 1481, 1583, 1585, 1587, 1589, 1591,
 1593, 1595, 1607, 1676, 1677, 1686, 1937,
 2447, 2449, 2451, 2453, 2881, 3184, 3194,
 3231, 3239, 3249, 3353, 3355, 3357, 3371,
 3482, 3484, 3490, 3573, 3671, 3697, 3711,
 3717, 3718
 See also strings; wide characters
 8-bit, reading 1677
 control 700, 1676, 1937
 disabling 700, 1937
 discarding on input 1676
 default (signed or unsigned) 51
 devices 3353, 3355, 3371
 input stream, injecting 3355
 size 3353, 3371
 escape 1686
 handling 2881
 international, See wide characters
 lowercase, converting to 1676, 3482
 multibyte 1583, 1585, 1587, 1589, 1591, 1593,
 1595, 3671, 3697, 3711, 3718
 bytes, counting 1583, 1585
 wide characters, conversion object 1589
 wide characters, converting to/from 1587, 1591,
 1593, 1595, 3671, 3697, 3711, 3718
 number waiting to be read 3357
 searching for 1607, 3184, 3239
 sets, searching for 3194, 3231, 3249
 special 1686

characters (*continued*)

- stdin, reading from 648, 827, 829
- stdout, writing to 707, 2451, 2453
- streams 645, 652, 704, 823, 825, 2447, 2449, 3573
 - pushing back 3573
 - reading from 645, 652, 823, 825
 - writing to 704, 2447, 2449
- testing for 1421, 1423, 1425, 1429, 1431, 1433, 1439, 1443, 1447, 1449, 1451, 1453, 1481
 - alphabetic 1423
 - alphanumeric 1421
 - ASCII 1425
 - blank 1429
 - control character 1431
 - decimal digit 1433
 - hexadecimal digit 1481
 - lowercase 1443
 - printable 1439, 1447
 - punctuation 1449
 - uppercase 1453
 - whitespace 1451
- uppercase, converting to 3484, 3490
- wide characters, converting to/from 238, 3717
 - See also strings; wide characters
- chdir() 286
- chmod() 289, 1225, 1234, 1237
 - resource managers, implementing in 1234, 1237
- chown() 292, 699, 1225, 1239, 1242, 1936
 - resource managers, implementing in 1239, 1242
 - restricting use of 699, 1936
- chroot() 295
- chsize() 298
- CIDR (Classless Internet Domain Routing) 1080, 1085
- CIOCCRYPT 1206
- CIOCGSESSION 1206
- CIOCKEY 1206
- CIS (Card Information Structure), reading 1961
- CLASS_APP 2945
- CLASS_DAEMON 2945
- CLASS_DISPLAY 2945
- CLASS_FSYS 2945
- CLASS_UNKNOWN 2945
- classes 1081, 1979, 3722
 - IP addresses 1081
 - PCI 1979
 - wide-character 3722
- Classless Internet Domain Routing (CIDR) 1080, 1085
- clearenv() 300
- clearerr() 302
- cli 1164
- ClientInfoExtFree() 304
- CLK_TCK 3472
- CLOCAL 3380
- clock 307, 310, 313, 319, 323, 325, 327, 329, 333, 2295, 3309, 3327
 - adjusting 323
 - CPU time, getting ID of for a thread 2295
 - cycles 325
 - getting 333
 - ID, getting 307, 327
 - period, getting and setting 329
 - resolution, getting 310

clock (*continued*)

- setting 319, 333
- ticks 3309, 3327
 - per seconds 3327
- time, getting 313
- clock_getcpuclockid() 307
- clock_getres() 310
- clock_gettime() 313
- CLOCK_HARMONIC 329
- CLOCK_MONOTONIC 309, 312, 315, 329, 1723, 1726, 2275, 2280, 2324, 2434, 2820, 3427, 3442, 3448, 3451, 3464
- clock_nanosleep() 316
- CLOCK_REALTIME 309, 312, 315, 319, 322, 329, 333, 2274, 2280, 2325, 2371, 2373, 2820, 3427, 3442, 3448, 3451, 3464
- clock_settime() 319, 2143
- CLOCK_SOFTTIME 309, 312, 315, 329, 2274, 2280, 3427, 3442, 3448, 3451, 3464
- clock_t 3309, 3471
- clock() 305
- ClockAdjust() 2143
- ClockAdjust(), ClockAdjust_r() 323
- ClockCycles() 325
- ClockId(), ClockId_r() 327
- ClockPeriod() 2142
- ClockPeriod(), ClockPeriod_r() 329
- CLOCKS_PER_SEC 305
- ClockTime() 2143
- ClockTime(), ClockTime_r() 333
- close() 336, 1249, 1251
 - resource managers, implementing in 1249, 1251
- closedir() 338, 1913
- closelog() 340
- cmsghdr 3578
- code, portability 46
- collating sequence, setting 2881
- COLUMNS 300
- commands 903, 977, 2491, 2668, 3330
 - executing 2491, 2668, 3330
 - on a remote host 2491, 2668
 - options, parsing 903, 977
- communications line 3340, 3362
 - break condition, asserting 3362
 - disconnecting 3340
- comparison 204, 1609, 1616, 3180, 3186, 3188, 3190, 3208, 3217, 3221, 3225, 3676, 3678, 3689, 3726
 - bytes 204, 1609, 1616
 - strings 1609, 1616, 3180, 3186, 3188, 3190, 3208, 3676, 3678, 3689, 3726
 - case-insensitive 1616, 3180, 3188, 3208
 - case-sensitive 1609, 3186
 - locale's collating sequence, using 3190
 - wide-character 3676, 3678, 3689, 3726
 - substrings 3217, 3221, 3225
 - case-insensitive 3217, 3225
 - case-sensitive 3221
- compiling with optimization 51
- complementary error function 534
- complex numbers, absolute value of 242
- computer, rebooting 3319

- condition variables 2257, 2259, 2261, 2263, 2265, 2269, 2272, 2274, 2276, 2278, 2281, 2282, 3280, 3282, 3289, 3305
 - attributes 2272, 2274, 2276, 2278, 2281, 2282
 - clock 2274, 2281
 - destroying 2272
 - initializing 2278
 - process-shared 2276, 2282
 - blocking on 2265, 2269
 - destroying 2259, 3289
 - initializing 2261, 3305
 - synchronization objects 3280, 3282
 - unblocking 2257, 2263
 - all threads 2257
 - highest priority thread 2263
 - waiting on 2265, 2269
 - timed 2265
 - configuration files, opening 267, 609
 - configuration strings 344, 883, 2868, 3547
 - _CS_HOSTNAME 883, 2868
 - _CS_TIMEZONE 3547
 - getting 344
 - setting 344
 - configuration values 3308
 - getting 3308
 - configuration values, getting 699, 1332, 1936
 - confstr() 344, 2143
 - connect functions (resource managers) 1271, 1325
 - default values, setting 1271
 - open, default 1325
 - connect() 348, 3359, 3766
 - ConnectAttach(), ConnectAttach_r() 352
 - ConnectClientInfo(), ConnectClientInfo_r() 356
 - ConnectClientInfoAble() 361
 - ConnectClientInfoExt() 363
 - ConnectDetach(), ConnectDetach_r() 281, 367
 - ConnectFlags() 2143
 - ConnectFlags(), ConnectFlags_r() 370
 - connections 58, 281, 304, 348, 352, 356, 361, 363, 367, 370, 372, 1523, 1630, 1869, 2480, 2494, 2656, 2942, 3309
 - client, information about 304, 356, 361, 363
 - detaching 281, 367
 - dispatch interface, creating 1630
 - flags, modifying 370
 - ID 352, 3309
 - resource constrain threshold 3309
 - server, information about 372
 - shutting down full-duplex 2942
 - sockets 58, 348, 1523, 1869, 2480, 2494, 2656
 - accepting on 58, 2480
 - initiating on 348, 1869, 2494
 - listening on 1523, 2656
 - ConnectServerInfo(), ConnectServerInfo_r() 280, 372
 - checking for stale _PULSE_CODE_COIDDEATH pulses 280
 - console I/O 3175, 3176, 3177
 - stderr 3175
 - stdin 3176
 - stdout 3177
 - const 45
 - contention scope 2201, 2225
 - control characters 700, 1431, 1461, 1676, 1937, 3382
 - disabling 700, 1937
 - discarding on input 1676
 - testing a character for 1431, 1461
 - controlling terminals 387, 3369
 - making 3369
 - path name 387
 - copysign(), copysignf() 375
 - core files, maximum size 2900
 - cos(), cosf(), cosl() 377
 - cosh(), coshf(), coshl() 379
 - cosines 120, 377, 379
 - hyperbolic 379
 - inverse hyperbolic 120
 - CPU 2759, 2761, 3320, 3322, 3517
 - index in trace events 3517
 - turning on or off 3320, 3322
 - usage 2759, 2761
 - overall, getting 2761
 - partition, getting for 2759
 - CPU-time clock ID 307, 327, 2295
 - for a process 307, 327
 - for a thread 327, 2295
 - CREAD 3380
 - creat(), creat64() 381, 3558
 - CRIOGET 1206
 - critical threads 2762, 2764, 2965
 - determining if 2764
 - marking as 2762, 2965
 - cross-endian support in resource managers 2583
 - crypt() 385
 - crypto 1206
 - CS5-CS8 3380
 - CSIZE 3380
 - CSTOPB 3381
 - ctermid() 387
 - ctime(), ctime_r() 389
 - CTL_MAXNAME 3311
 - CTL_NET 3312
 - cube roots 258
 - current working directory 286, 610, 830, 997
 - curses 1873
- ## D
- D_FLAG_FILTER 419
 - D_FLAG_STAT 420
 - D_GETFLAG 419
 - D_SETFLAG 419
 - daemon() 394
 - daemons 394, 2156, 2159, 2164
 - system 394, 2156, 2159, 2164
 - termination, notification of 2159, 2164
 - data segment 213, 491, 494, 2900
 - changing space allocated for 213
 - end of 491, 494
 - maximum size 2900
 - data server 469, 472, 474, 476, 478, 480, 481
 - applications 474, 480
 - deregistering 474
 - registering 480

- data server (*continued*)
 - variables 469, 472, 476, 478, 481
 - creating 472
 - deleting 469
 - flags, setting 476
 - getting 478
 - setting 481
- data streams, flow control 3342
- databases 385, 496, 497, 522, 523, 524, 525, 526, 654, 851, 853, 855, 857, 859, 866, 870, 872, 876, 878, 881, 898, 900, 902, 924, 926, 928, 930, 932, 934, 937, 938, 941, 952, 954, 956, 971, 975, 1012, 1017, 1019, 1024, 1131, 1876, 2184, 2459, 2474, 2507, 2663, 2842, 2862, 2866, 2884, 2892, 2894, 2905, 2912, 3743
- blocks 2507, 3743
 - reading 2507
 - writing 3743
- groups 496, 851, 853, 855, 857, 859, 1131, 2862
 - closing 496
 - ID, getting information about 853, 855
 - membership 1131
 - name, getting information about 857, 859
 - next entry, getting 851
 - rewinding 2862
- hosts 497, 866, 870, 872, 876, 878, 881, 1012, 1017, 1019, 1024, 2866
 - closing 497
 - entries, getting 866, 870, 872, 876, 878, 881
 - errors 1012, 1017, 1024
 - opening 2866
 - structure 1019
- network 522, 898, 900, 902, 1876, 2884
 - closing 522
 - entries, getting 898, 900, 902
 - opening 2884
 - structure 1876
- passwords 385, 524, 930, 932, 934, 937, 938, 941, 2474, 2894
 - closing 524
 - encrypting 385, 2474
 - entry, getting for a user 934, 937, 938, 941
 - entry, getting next 930, 932
 - rewinding 2894
- protocols 523, 924, 926, 928, 2184, 2892
 - closing 523
 - entries, getting 924, 926, 928
 - opening 2892
 - protoent 2184
- services 525, 952, 954, 956, 2842, 2905
 - closing 525
 - entries, getting 952, 954, 956
 - entry structure 2842
 - opening 2905
- shadow passwords 526, 654, 971, 975, 2459, 2912
 - closing 526
 - entry, reading 654, 971, 975
 - entry, structure 2459
 - entry, writing 2459
 - rewinding 2912
- system packet forwarding, See ROUTE
- datagrams 3075, 3076, 3553
- date 3547
- daylight 396, 3547
- daylight saving time 396, 3547
- DCMD_ALL_GETFLAGS 1199, 1206, 1257
- DCMD_ALL_GETMOUNTFLAGS 1257, 1692
- DCMD_ALL_SETFLAGS 1199, 1206, 1257
- DCMD_CHR_GETSIZE 1199
- DCMD_CHR_ISCHARS 1199
- DCMD_CHR_LINESTATUS 1199
- DCMD_CHR_OSCHARS 1199
- DCMD_CHR_SERCTL 1199
- DCMD_CHR_SETSIZE 1199
- DCMD_CHR_TCDRAIN 1199
- DCMD_CHR_TCFLOW 1199
- DCMD_CHR_TCFLUSH 1199
- DCMD_CHR_TCGETATTR 1199
- DCMD_CHR_TCGETPGRP 1199
- DCMD_CHR_TCINJECTC 1199
- DCMD_CHR_TCSETATTR 1199
- DCMD_CHR_TCSETATTRD 1199
- DCMD_CHR_TCSETATTRF 1199
- DCMD_CHR_TCSETPGRP 1199
- DCMD_PROC_MAPINFO 1662
- DCMD_PROC_PAGEDATA 1662
- DEAD_PROCESS 987, 3595
- DebugBreak() 397
- debugging 397, 398, 399, 459, 965, 2043, 2563, 3094
 - kernel 398, 399
 - printing debugging messages (resolver routines) 2563
 - processes 397, 2043, 3094
 - shared objects 459
 - sockets 965
- DebugKDBreak() 398
- DebugKDOutput() 399
- decimal digit, testing a character for 1433, 1465
- decimal-point character, setting 2881
- delay() 401
- DELAYTIMER_MAX 3435
- dev_t 2689
- devctl() 403, 1206, 1257, 1261, 1264
 - resource managers, implementing in 1206, 1257, 1261, 1264
- devctlv() 403
- DEVDIR_FROM 406
- DEVDIR_NONE 406
- DEVDIR_TO 406
- DEVDIR_TOFROM 406
- devices 403, 1198, 1666, 1672, 1677, 1680, 1686, 1692, 1834, 1966, 1977, 1979, 1982, 1989, 1991, 1993, 1995, 1997, 1999, 2001, 2004, 2006, 2507, 2509, 2603, 2688, 2689, 2691, 3167, 3338, 3353, 3355, 3369, 3371, 3743
 - block special 2507, 3743
 - reading 2507
 - writing 3743
 - character 3353, 3355, 3371
 - characters, injecting 3355
 - size 3353, 3371
 - classes of 2689
 - controlling 403, 1198
 - controlling device, making 3369
 - I/O memory, mapping 1666, 1834

devices (*continued*)

- mounts, autodetecting 1692
- numbers 2603, 2688, 2691, 3167
 - attaching 2688
 - detaching 2691
 - getting 2603
 - manipulating 3167
- opening 1672
- output, waiting for completion 3338
- PCI 1966, 1977, 1979, 1982, 1989, 1991, 1993, 1995, 1997, 1999, 2001, 2004, 2006
 - attaching 1966
 - configuration, reading 1989, 1991, 1993, 1995
 - configuration, writing 1999, 2001, 2004, 2006
 - detaching 1977
 - finding 1979, 1982
 - rescanning for 1997
- reading 1677, 2509
- script, running on 1680
- writing 1686
- devp-pccard server 1950, 1952, 1954, 1956, 1958, 1961, 1962
 - arming 1950
 - attaching 1952
 - card insertion/removal, notification of 1950
 - CIS (Card Information Structure), reading 1961
 - detaching 1954
 - locking 1958
 - socket setup information 1956
 - unlocking 1962
- difftime() 417
- digit, testing a character for 1433, 1465, 1479, 1481
 - decimal 1433, 1465
 - hexadecimal 1479, 1481
- DIOCGGETSRCNODES 1206
- DIOCGGETSTATES 1207
- DIOCGGETIFACES 1207
- DIOCRADDADDRS 1207
- DIOCRADDTABLES 1207
- DIOCRCLRASTATS 1207
- DIOCRCLRTSTATS 1207
- DIOCRDELADDRS 1207
- DIOCRDELTABLES 1207
- DIOCRGETADDRS 1207
- DIOCRGETASTATS 1207
- DIOCRGETTABLES 1207
- DIOCRGETTSTATS 1207
- DIOCRINADEFINE 1207
- DIOCRSETADDRS 1207
- DIOCRSETTFLAGS 1207
- DIOCRTSTADDRS 1207
- DIOCXBEGIN 1207
- DIOXC_COMMIT 1207
- DIOCXROLLBACK 1207
- DIR 2514
- dirctl() 419
- direct 149, 2711
- Direct Memory Access (DMA) 250, 2674
 - cache coherency 250
 - channels, managing 2674
- directories 61, 149, 202, 286, 295, 338, 419, 420, 422, 424, 488, 610, 700, 798, 830, 997, 1636, 1642,

directories (*continued*)

- 1888, 1913, 1937, 1940, 2156, 2514, 2518, 2648, 2658, 2712, 2789, 3375, 3661
- access, checking 61, 488
- base name 202
- closing 338
- controlling 419
- creating 1636, 1642
- current working 286, 610, 830, 997, 2156
 - for daemons 2156
- deleting 2658
- entries 149, 419, 422
 - duplicate, filtering 419
 - information about 422
 - sorting 149
- files, searching for 1940
- hierarchy, walking 798, 1888
- information, requesting 420
- name 424
- opening 1913
- position 2789, 3375
 - getting 3375
 - setting 2789
- reading 2514, 2518
- rewinding 2648
- root 295
- scanning 2712
- unlinking, support for (_PC_LINK_DIR) 700, 1937
- waiting for 3661
- dirent 2514, 2518
- dirent_extra, dirent_extra_stat 2515
- dirent, dirent64 422
- dirname() 424
- dispatch interface 426, 429, 431, 433, 436, 438, 440, 443, 445, 1625, 1626, 1630, 1633, 1842, 1848, 1850, 1852, 2443, 2445, 2581, 2600, 2797, 2800, 2803, 3386, 3387, 3393, 3396, 3397
- See also resource managers
- blocking 426
- connections, creating 1630
- context 429, 431
 - allocating 429
 - freeing 431
- endian, converting 1626
- events 440, 2803
 - handling 440
 - last selected 2803
- file descriptors 2797, 2800
 - handle, attaching 2797
 - handle, detaching 2800
- handles 433, 436, 438
 - creating 433, 436
 - destroying 438
- message handlers 1625, 1633
 - attaching 1625
 - detaching 1633
- names 1842, 1848, 1850, 1852
 - attaching 1842
 - detaching 1850
 - server connections, closing 1848
 - server connections, opening 1852

- dispatch interface (*continued*)
 - path 2581, 2600
 - attaching to 2581
 - detaching from 2600
 - pulse handlers 2443, 2445
 - attaching 2443
 - detaching 2445
 - thread pool 3386, 3387, 3393, 3396, 3397
 - attributes, changing 3386, 3396
 - creating 3387
 - destroying 3393
 - starting 3397
 - timeout, setting 443
 - unblocking 445
 - See also resource managers
- dispatch_block() 426
- dispatch_context_alloc() 429
- dispatch_context_free() 431
- dispatch_create_channel() 436
- dispatch_create() 433
- dispatch_destroy() 438
- dispatch_handler() 440
- dispatch_t 434, 437
- dispatch_timeout() 443
- dispatch_unblock() 445
- DISPLAY_CLASS 2945
- DISPLAY_PROC 2945
- display, controlling a terminal's 1873
- DisplayData_t 2946
- div_t 447
- div() 447
- division 447, 1508
 - integer 447
 - long and long long integers 1508
- DL_DEBUG 459
- DI_info 449
- dladdr() 449
- dlclose() 451
- dLError() 453
- dlopen() 455, 2146
- dlsym() 461
- DMA (Direct Memory Access) 250, 2674
 - cache coherency 250
 - channels, managing 2674
- dn_comp() 465
- dn_expand() 466
- domains 345, 465, 466, 833, 913, 1012, 1017, 1024, 2563, 2567, 2570, 2573, 2576, 2578, 2849, 3577
 - errors 1012, 1017, 1024
 - names 345, 465, 466, 833, 2563, 2567, 2570, 2573, 2576, 2578, 2849
 - compressing 465
 - expanding 466
 - getting 345, 833
 - resolving 2563, 2567, 2570, 2573, 2576, 2578
 - setting 2849
 - secure RPC 345
 - UNIX 913, 3577
 - peer credentials 913
- dot notation (IP addresses) 1081
- dotted quad (IP addresses) 1081
- double-precision numbers 118, 120, 157, 159, 163, 165, 167, 171, 242, 258, 260, 375, 377, 379, 417, 468, 531, 532, 534, 601, 603, 606, 671, 675, 695, 725, 816, 1040, 1057, 1437, 1441, 1445, 1486, 1488, 1490, 1506, 1513, 1539, 1541, 1543, 1545, 1689, 1885, 2129, 2133, 2554, 2654, 2661, 2705, 2707, 2713, 2979, 3024, 3026, 3153, 3253, 3334, 3336, 3703, 3754, 3756, 3758
 - absolute value 242, 606
 - arccosines 118
 - arcsines 157
 - arctangents 163, 165
 - Bessel functions 1486, 1488, 1490, 3754, 3756, 3758
 - complementary error function 534
 - cosines 377
 - cube roots 258
 - error function 532
 - exceptions, signal for 2979
 - exponentials 601, 603
 - exponents, radix-independent 2705, 2707
 - finite, determining if 1437
 - fractional part of a double-precision number 1689
 - gamma functions 816, 1513
 - hyperbolic cosines 379
 - hyperbolic sines 3026
 - hyperbolic tangents 3336
 - hypotenuse, length of 1040
 - infinite, determining if 1441
 - input, formatted 2713
 - integral logarithms 1057
 - integral part of a double-precision number 1689
 - integral power of 2 725, 1506
 - inverse hyperbolic cosines 120
 - inverse hyperbolic sines 159
 - inverse hyperbolic tangents 167
 - logarithms 1539, 1541, 1543
 - modular arithmetic 675
 - next representable 1885
 - normalized fractions 725
 - not a number, determining if 1445
 - powers 2129
 - precision 695
 - printing 2133
 - pseudo-random numbers 468, 531
 - radix-independent exponents 1545, 2705, 2707
 - remainders 2554
 - residue 675
 - rounding 260, 671, 2654, 2661
 - sign, copying 375
 - sines 3024
 - square roots 3153
 - strings, converting to/from 171, 3253
 - tangents 3334
 - times, difference between 417
 - wide-character strings, converting to/from 3703
- doubly linked lists 1145, 2559
 - inserting elements 1145
 - removing elements 2559
- drand48() 468
- ds_clear() 469
- ds_create() 472

- ds_deregister() 474
- ds_flags() 476
- ds_get() 478
- DS_PERM 471, 474, 476
- ds_register() 480
- ds_set() 481
- dup() 483
- dup2() 485
- duplex connection, shutting down 2942
- dynamic rules filter, controlling 3516
- dynamically linked libraries 449, 451, 453, 455, 459, 461
 - addresses, translating 449
 - closing 451
 - debugging 459
 - errors 453
 - opening 455
 - symbol, getting address of 461

E

- E2BIG 540
- EACCES 540
- eaccess() 488
- EADDRINUSE 540
- EADDRNOTAVAIL 540
- EADV 540
- EAFNOSUPPORT 540, 3079
- EAGAIN 540
- EALREADY 540, 547
 - changes to the value of 547
- EALREADY_DYNAMIC 547
- EALREADY_NEW 547
- EALREADY_OLD 547
- EBADE 540
- EBADF 540, 1950, 1959, 1963
- EBADFD 540
- EBADFSYS 540
- EBADMSG 540
- EBADR 540
- EBADRPC 540
- EBADRQC 540
- EBADSLT 540
- EBFONT 540
- EBUSY 540, 1959
- ECANCELED 540
- ECHILD 540
- ECHO 3381
- ECHOE 3381
- ECHOK 3381
- ECHONL 3381
- ECHRNG 540
- ECOMM 540
- ECONNABORTED 540
- ECONNREFUSED 540, 1523
- ECONNRESET 540
- ECTRLTERM 540
- EDEADLK 540
- EDEADLOCK 540
- EDESTADDRREQ 540
- EDOM 540
- EDQUOT 540
- EENDIAN 540
- EEXIST 540
- EFAULT 540
- EFBIG 540
- EFPOS 540
- EHOSTDOWN 540
- EHOSTUNREACH 540
- EIDRM 540
- EILSEQ 540
- EINPROGRESS 540
- EINTR 540
- EINVAL 540
- EIO 540
- EISCONN 540, 3766
- EISDIR 540
- EL2HLT 540
- EL2NSYNC 540
- EL3HLT 540
- EL3RST 540
- elements 1145, 2559
 - inserting into a list 1145
 - removing from a list 2559
- ELIBACC 540
- ELIBBAD 540
- ELIBEXEC 540
- ELIBMAX 540
- ELIBSCN 540
- ELNRNG 540
- ELOOP 540
- EMFILE 540
- EMLINK 540
- EMORE 540
- EMPTY 3594
- EMSGSIZE 540
- EMULTIHOP 540
- ENAMETOOLONG 540
- Encapsulated Security Payload (ESP) 1396
- encrypt() 492
- encrypted filesystems 727, 729, 732, 735, 738, 739, 741, 743, 745, 748, 750, 753, 755, 757, 760, 762, 764, 766, 769
 - checking support 727
 - domains 729, 732, 735, 739, 741, 743, 745
 - creating 729
 - destroying 743
 - key, changing 732
 - key, checking 735
 - locking 739
 - querying 741
 - unlocking 745
 - enabling 748, 750
 - file domain 753, 766
 - getting 753
 - setting 766
 - files 764
 - migration 764
 - key size 738
 - keys 755
 - generating 755
 - logging 769
 - migrating 762
 - migration 757, 760

encryption 385, 492, 2474, 2877
 key, setting 2877
 passwords 385, 2474
 strings 492
 end-of-file 302, 529, 637
 files 529
 streams 302, 637
 clearing 302
 endsent() 495
 endgrent() 496
 endhostent() 497
 endian 51, 498, 500, 502, 504, 506, 508, 510, 512, 514,
 516, 518, 520, 1063, 1067, 1626, 1927, 1931,
 2583, 3270, 3561, 3563, 3565, 3567, 3568,
 3569
 big 51, 498, 500, 502, 510, 512, 514, 516, 518, 520,
 1626, 3270, 3561, 3563, 3565, 3567, 3568,
 3569
 __BIGENDIAN__ manifest 51
 little endian, converting to/from 510, 512, 514, 516,
 518, 520, 3270
 messages 1626
 native format, converting to/from 498, 500, 502
 unaligned values, accessing safely 3567, 3568, 3569
 unaligned values, writing safely 3561, 3563, 3565
 cross-endian support in resource managers 2583
 little 51, 504, 506, 508, 510, 512, 514, 516, 518,
 520, 1626, 3270, 3561, 3563, 3565, 3567,
 3568, 3569
 __LITTLEENDIAN__ manifest 51
 big endian, converting to/from 510, 512, 514, 516,
 518, 520, 3270
 messages 1626
 native format, converting to/from 504, 506, 508
 unaligned values, accessing safely 3567, 3568, 3569
 unaligned values, writing safely 3561, 3563, 3565
 ports 1063, 1067, 1927, 1931
 reading from 1063, 1067
 writing to 1927, 1931
 ENDIAN_BE16() 498
 ENDIAN_BE32() 500
 ENDIAN_BE64() 502
 ENDIAN_LE16() 504
 ENDIAN_LE32() 506
 ENDIAN_LE64() 508
 ENDIAN_RET16() 510
 ENDIAN_RET32() 512
 ENDIAN_RET64() 514
 ENDIAN_SWAP16() 516
 ENDIAN_SWAP32() 518
 ENDIAN_SWAP64() 520
 endnetent() 522
 endprotoent() 523
 endpwent() 524
 endservent() 525
 endspent() 526
 endutent() 527
 ENETDOWN 540
 ENETRESET 540
 ENETUNREACH 540
 ENFILE 540
 ENOANO 540
 ENOBUFS 540, 2664
 ENOCSI 540
 ENODATA 540
 ENODEV 540, 1957, 1959, 1963
 ENOENT 540
 ENOEXEC 540
 ENOLCK 540
 ENOLIC 540
 ENOLINK 540
 ENOMEM 540
 ENOMSG 540
 ENONDP 540
 ENONET 540
 ENOPKG 540
 ENOPROTOOPT 540
 ENOREMOTE 540
 ENOSPC 540
 ENOSR 540
 ENOSTR 540
 ENOSYS 540
 ENOTBLK 540
 ENOTCONN 540
 ENOTDIR 540
 ENOTEMPTY 540
 ENOTSOCK 540
 ENOTSUP 540
 ENOTTY 540
 ENOTUNIQ 540
 ENTER 1021
 ENTRY 1021
 env 570, 592, 2853, 3101, 3113, 3130, 3142
 environ 528, 1566, 2854
 environment 1549, 2875, 2970, 3010
 restoring 1549, 2970
 saving 2875, 3010
 environment variables 152, 247, 254, 255, 256, 300, 455,
 458, 459, 460, 528, 569, 592, 838, 873, 883,
 1569, 1570, 1571, 1572, 1573, 1604, 2019,
 2034, 2038, 2454, 2527, 2528, 2529, 2563,
 2568, 2571, 2573, 2576, 2579, 2785, 2853,
 2868, 2921, 3090, 3106, 3111, 3116, 3117,
 3122, 3123, 3135, 3140, 3145, 3146, 3330,
 3547, 3584
 CACHE_MSYNC 247, 254
 CACHE_NOP 247, 254
 COLUMNS 300
 defining 528, 569, 592, 2038, 2454, 2853, 3090,
 3106, 3117, 3123, 3135, 3146
 deleting 300, 2038, 2454, 2853, 3090, 3106, 3117,
 3123, 3135, 3146, 3584
 DL_DEBUG 459
 files, searching for 2785
 getting 838
 HOSTALIASES 873
 HOSTNAME 883, 2868
 INCLUDE 2785
 LD_BIND_NOW 459
 LD_DEBUG 460
 LD_DEBUG_OUTPUT 460
 LD_LIBRARY_PATH 455
 LD_PRELOAD 458, 460
 LIB 2785

environment variables (*continued*)

- LINES 300
- LOCALDOMAIN 2563, 2568, 2571, 2573, 2576, 2579
- MALLOC_ARENA_CACHE_MAXBLK 1570
- MALLOC_ARENA_CACHE_MAXSZ 1570
- MALLOC_ARENA_SIZE 152, 1571
- MALLOC_BAND_CONFIG_STR 1572
- MALLOC_BTDEPTH 1573
- MALLOC_CHKACCESS_LEVEL 1573
- MALLOC_DUMP_LEAKS 1573
- MALLOC_FREE_LIFO 1571
- MALLOC_INITVERBOSE 1573
- MALLOC_MAX_ALIGNMENT 1604, 2034
- MALLOC_MEMORY_HOLD 1571
- MALLOC_MEMORY_PREALLOCATE 1572
- MALLOC_MMAP_NOZERO 1572
- MALLOC_OPTIONS 255, 256, 1569, 1572, 2527, 2528, 2529
- MALLOC_TRACE 1573
- MALLOC_TRACEBT 1573
- PATH 300, 569, 592, 2785, 3111, 3116, 3122, 3140, 3145
- pointer to 528
- SHELL 300, 2019, 3330
- STDIO_DEFAULT_BUFSIZE 2921
- TERM 300
- TERMINFO 300
- TZ 300, 3547

ENXIO 540

EOF 3615

eof() 529

EOK 540

EOPNOTSUPP 540, 3079

EOVERFLOW 540

EOWNERDEAD 540

EPERM 540

EPFNOSUPPORT 540

EPIPE 540

EPROCUNAVAIL 540

EPROGMISMATCH 540

EPROGUNAVAIL 540

EPROTO 540

EPROTONOSUPPORT 540, 3079

EPROTOTYPE 540

erand48() 531

ERANGE 540

EREMCHG 540

EREMOTE 540

ERESTART 540, 1743

erf(), erff() 532

erfc() 534

erfcf() 534

erfcl() 534

EROFS 540

ERPCMISMATCH 540

err(), errx() 536

errno 539

- threads 539

error function 532, 534

- complementary 534

error messages, IPsec policy manipulation library 1410

errors 302, 536, 539, 637, 639, 814, 904, 1012, 1017, 1024, 1743, 2009, 2482, 2547, 2974, 3175, 3198, 3200, 3612

See also signals

command-line options, printing for 904

end-of-file 637

errno global variable 539

hosts 1012, 1017, 1024

message-passing 1743

messages 536, 2009, 3198, 3200, 3612

- for an error code 2009, 3198, 3200
- formatted 536, 3612

regular expressions 2547

resolver 1012, 1017, 1024

signals, raising 2482

socket address information 814

stderr 536, 2009, 3175, 3612

stream I/O 302, 639

- clearing 302
- testing for 639

See also signals

escape characters 1686

ESHUTDOWN 540

ESOCKTNOSUPPORT 540

ESP (Encapsulated Security Payload) 1396

ESPIPE 540

ESRCH 540

ESRMNT 540

ESRVRFAULT 540

ESTALE 540

ESTRPIPE 540

ETIME 540

ETIMEDOUT 540

ETOOMANYREFS 540

ETXTBSY 540

EUNATCH 540

EUSERS 540

event_data_t 3516

events 426, 440, 445, 1154, 1162, 1736, 1817, 2158, 2163, 2168, 2170, 2178, 2182, 2803, 2961

- blocking while waiting for 426
- channels, delivering through 1736
- checking validity of 1817
- handling 440
- interrupts 1154, 1162
 - attaching 1154
 - detaching 1162
- last selected 2803
- sigevent 2182, 2961
 - triggering 2182
 - type, determining with SIGEV_GET_TYPE() 2961
- system 2158, 2163, 2168, 2170, 2178
 - notification of 2158, 2163, 2168, 2178
 - triggering 2170
- unblocking 445

events (trace) 3492, 3494, 3496, 3498, 3500, 3502, 3504, 3506, 3508, 3510, 3517, 3519, 3524, 3526, 3529, 3531, 3533, 3534, 3537

callbacks, attaching 3526, 3529

class and event type 3517

controlling the tracing of 3510

CPU index 3517

events (trace) (*continued*)

- debug mode, setting 3531
- inserting 3492, 3494, 3496, 3498, 3500, 3502, 3504, 3506, 3508
 - arbitrary class and type 3500
 - current address 3496
 - function entry 3492
 - function exit 3494
 - user combine event 3498
 - user simple event 3504
 - user string event 3502, 3506, 3508
- parser state structure 3533, 3534, 3537
 - destroying 3533
 - information about 3534
 - initializing 3537
- parsing 3524
- user-generated 3519
- EWOLDBLOCK 540, 967
- exceptional conditions, file descriptors with 2673, 2792
- exceptions, floating-point 690, 693
 - mask 690
 - registers 693
- exclusive locks 666
- EXDEV 540
- exec* family of functions 1913
- execl() 550
- execle() 556
- execlp() 562
- execlpe() 569
- executable files 341, 342, 784, 1691, 2183, 3171
 - base name 2183
 - file descriptor 341
 - full path 342
 - mounted filesystem, preventing from loading on 784, 1691, 3171
- execv() 573
- execve() 579
- execvp() 585
- execvpe() 592
- EXFULL 540
- exit() 599
- exp(), expf(), expl() 601
- expm1(), expm1f() 603
- exponentials, floating point 601, 603, 725, 1506
- exponents, radix-independent 1545, 2705, 2707
- export 570, 592, 2853, 3101, 3113, 3130, 3142
- Extended Message Signaled Interrupts (MSI-X) 1970

F

- F_ALLOCSP 623, 1351
- F_ALLOCSP64 624
- F_DUPFD 624
- F_FREESP 624, 1351
- F_FREESP64 624
- F_GETFD 624
- F_GETFL 624, 1199
- F_GETLK 624
- F_GETLK64 625
- F_GETOWN 1199
- F_LOCK 1535
- F_OK 61, 488

- F_RDLCK 625
- F_SETFD 625, 1199
- F_SETFL 625, 1199
- F_SETLK 625, 1199
- F_SETLK64 626
- F_SETLKW 626
- F_SETLKW64 626
- F_SETOWN 1199
- F_TEST 1535
- F_TLOCK 1535
- F_ULOCK 1535
- F_UNLCK 625
- F_WRLCK 625
- fabs(), fabsf() 606
- fast mode, controlling 3514
- fcfgopen() 609
- fchdir() 610
- fchmod() 612
- fchown() 615
- FCHR_MAX 2903
- fclose() 618
- fcloseall() 621
- fcntl() 483, 623, 1281
 - F_DUPFD 483
 - resource managers, implementing in 1281
- FD, See file descriptors
- FD_CLOEXEC 625, 1199, 2040, 2936, 3089, 3122
- FD_CLR() 2793
- FD_ISSET() 2793
- FD_SET() 2793
- FD_SETSIZE 2793
- FD_ZERO() 2793
- fdatasync() 631
- fdinfo() 1267, 1268
 - resource managers, implementing in 1267, 1268
- fdistrusted() 633
- fdopen() 635
- feof() 637
- ferror() 639
- fflush() 641
- ffs() 644
- fgetc() 645
- fgetchar() 648
- fgetpos() 650
- fgets() 652, 950
- fgetspent() 654
- fgetwc() 656
- fgetws() 658
- FIFO scheduling 2223
- FIFOs 1639, 1642, 2007, 2011, 2019, 2131, 2502, 2523
 - creating 1639, 1642
 - reading from 2131, 2502, 2523
 - unnamed (pipes) 2007, 2011, 2019
 - closing 2007
 - creating 2011
 - opening 2019
- file descriptors 352, 483, 485, 624, 635, 661, 835, 1427, 1435, 1905, 1916, 2013, 2040, 2048, 2050, 2052, 2673, 2792, 2793, 2797, 2800, 2843,

- file descriptors (*continued*)
 - 2900, 3084, 3088, 3089, 3092, 3093, 3122, 3175, 3176, 3177, 3542, 3544
- child process 2040, 2048, 2050, 2052, 3089, 3092, 3093, 3122
 - closing in 2040, 2048, 3089, 3093, 3122
 - specifying in 2040, 2050, 2052, 3089, 3092, 3122
- connection IDs as 352
- creating 1905, 3084
- duplicating 483, 485, 624, 1916, 2040, 2050, 3088
- exceptional conditions 2673, 2792
- full path 3542, 3544
- handle 2797, 2800
 - attaching 2797
 - detaching 2800
- maximum number of 2900
- polling 2013
- properties 1435
- ready for reading or writing 2673, 2792
- selecting 2673, 2792
- sets of, manipulating 2793
- setting the minimum acceptable 2843
- stderr 3175
- stdin 3176
- stdout 3177
- streams, associating with 635, 661
- table size, getting 835
- terminal, testing for association with 1427

fileno() 661

files 61, 202, 267, 289, 292, 298, 336, 341, 342, 381, 422, 424, 488, 527, 609, 612, 615, 623, 624, 625, 631, 633, 641, 650, 663, 666, 669, 673, 678, 681, 699, 700, 702, 704, 707, 709, 711, 713, 721, 770, 772, 776, 779, 784, 787, 793, 796, 798, 801, 802, 806, 810, 985, 988, 989, 1000, 1004, 1136, 1137, 1139, 1141, 1143, 1191, 1198, 1322, 1332, 1364, 1375, 1378, 1502, 1516, 1535, 1556, 1558, 1563, 1645, 1647, 1691, 1888, 1905, 1916, 1936, 1937, 1940, 1943, 1945, 2022, 2025, 2040, 2048, 2052, 2131, 2156, 2183, 2463, 2470, 2496, 2502, 2506, 2520, 2523, 2556, 2560, 2646, 2712, 2785, 2900, 2919, 3084, 3088, 3164, 3171, 3276, 3309, 3357, 3373, 3377, 3478, 3480, 3539, 3542, 3544, 3558, 3579, 3588, 3591, 3596, 3616, 3618, 3620, 3622, 3661, 3737, 3742, 3745

_PATH_UTMP 527, 985, 988, 989, 2463, 2919, 3596

access times 784, 1364, 1375, 1378, 1691, 3171, 3591

- disabling logging of on mounted filesystems 784, 1691, 3171
- resource managers, implementing in 1364, 1375, 1378

access, checking 61, 488

advisory information 2022

asynchronous I/O, support for (_PC_ASYNC_IO) 699, 1936

base name 202

closing 336, 2040, 2048, 2052

configurable limits 699, 1332, 1936

configuration, opening 267, 609

controlling 623

files (*continued*)

core, maximum size 2900

creating 381, 681, 784, 1191, 1322, 1691, 1905, 3084, 3171

- low-level 381
- not allowing on mounted filesystems 784, 1691, 3171
- resource manager 1191, 1322

deleting 2556, 3579

device parameters, manipulating 1198

directory entries for 422

directory name 424

executable 341, 342, 2183

- base name 2183
- file descriptor 341
- full path 342

extending 298, 623

flushing 641, 673

- all 673

information, getting 779, 1558, 3164

input, formatted 770, 810, 3618, 3622

link count, maximum 700, 1937

linking to 663, 1516, 1943, 1945, 2520, 3276

locking 624, 666, 669, 796, 1535

- by a thread 669
- nonblocking 796

modification times 802, 1364, 1375, 1378, 3588, 3591

- resource managers, implementing in 1364, 1375, 1378

monitoring changes to 1136, 1137, 1139, 1141

names 678, 700, 1937, 3542, 3544

- full path 3542, 3544
- matching 678
- maximum length 700, 1937

opening 381, 681, 1322, 1905, 2040, 3084

- resource manager 1322

output, formatted 702, 806, 3616, 3620

ownership, changing 292, 615, 1502

pathnames matching a pattern 1000, 1004

permissions 289, 612, 699, 1936, 2156, 3558

- changing 289, 612
- daemons 2156
- on creation 3558
- restricting the changing of 699, 1936

position 650, 772, 776, 1556, 3373

- getting 650
- setting 772, 776, 1556, 3373

prioritized I/O, support for (_PC_ASYNC_IO) 700, 1937

private access 1916

processes, maximum files per 3309

reading 1143, 2131, 2496, 2502, 2506, 2523, 3357

- blocks 2506
- characters, number waiting to be read 3357
- checking 2496
- iov_t 2523

renaming 2560

reopening 721

rewinding 2646

scanning directories for 2712

searching 1940, 2785

- environment variables 2785
- list of directories 1940

seeking 772

- files (*continued*)
 - shared access 3084, 3088
 - size, changing 298
 - space, allocating 2025
 - status flags 624
 - status-change times 1364, 1375, 1378
 - resource managers, implementing in 1364, 1375, 1378
 - synchronizing 631, 787
 - synchronous I/O, support for (`_PC_ASYNC_IO`) 700, 1937
 - temporary 1645, 1647, 3377, 3478, 3480
 - creating 3478
 - creating and opening 1645
 - name, generating 1647, 3377, 3480
 - tree, walking 798, 1888
 - truncating 298, 624, 793, 1563, 3539
 - trusted 633
 - unlinking 2556, 3579
 - unlocking 625, 666, 801, 1535
 - waiting for 3661
 - writing 704, 707, 709, 711, 713, 2470, 3737, 3742, 3745
 - blocks 3742
 - characters 704, 707
 - `iov_t` 3745
 - strings 709
 - wide characters 711
 - wide-character strings 713
- filesystems 495, 727, 729, 732, 735, 738, 739, 741, 743, 745, 748, 750, 753, 755, 757, 760, 762, 764, 766, 769, 783, 784, 842, 844, 847, 1691, 1692, 1694, 2860, 3170, 3171, 3279, 3559
 - encrypted 727, 729, 732, 735, 738, 739, 741, 743, 745, 748, 750, 753, 755, 757, 760, 762, 764, 766, 769
 - checking support 727
 - domains 729, 732, 735, 739, 741, 743, 745
 - enabling 748, 750
 - files 753, 764, 766
 - key size 738
 - keys 755
 - login 769
 - migrating 762
 - migration 757, 760
 - free space 784, 3171
 - information, getting 783, 3170
 - mounting 1692, 1694
 - options, parsing 1694
 - read-only 784, 1691, 3171
 - synchronizing 3279
 - table 495, 842, 844, 847, 2860
 - closing 495
 - getting next entry 844
 - opening 2860
 - searching for a block special device 847
 - searching for a filesystem 842
 - unmounting 1691, 3559
- filters (trace event), controlling 3515, 3516
- FIND 1021
- finite number, determining if 1437
- FIOASYNC 1199
- FIOCLEX 1199
- FIOGETOWN 1199
- FIONBIO 1199
- FIONCLEX 1199
- FIONREAD 1199
- FIOSETOWN 1199
- first-in first-out scheduling 2223
- FLAG_DEBUG 2944
- FLAG_FAST 2944
- FLAG_NO_DAEMON 2944
- FLAG_NONROOT 2944
- FLAG_UNATTENDED 2944
- FLAG_VERBOSE 2944
- FLAG_VERY_VERBOSE 2944
- flink() 663
- floating point 50, 118, 120, 157, 159, 163, 165, 167, 242, 258, 260, 375, 377, 379, 532, 534, 601, 603, 606, 671, 675, 690, 693, 695, 697, 725, 816, 1040, 1057, 1437, 1441, 1445, 1486, 1488, 1490, 1506, 1513, 1539, 1541, 1543, 1545, 1689, 1885, 2129, 2133, 2554, 2654, 2661, 2705, 2707, 2713, 2979, 3024, 3026, 3153, 3334, 3336, 3703, 3754, 3756, 3758
 - absolute value 242, 606
 - arccosines 118
 - arcsines 157
 - arctangents 163, 165
 - Bessel functions 1486, 1488, 1490, 3754, 3756, 3758
 - complementary error function 534
 - cosines 377
 - cube roots 258
 - error function 532
 - exceptions, signal for 2979
 - exponentials 601, 603
 - exponents, radix-independent 2705, 2707
 - finite, determining if 1437
 - fractional part 1689
 - gamma functions 816, 1513
 - hyperbolic cosines 379
 - hyperbolic sines 3026
 - hyperbolic tangents 3336
 - hypotenuse, length of 1040
 - infinite, determining if 1441
 - input, formatted 2713
 - integral logarithms 1057
 - integral part 1689
 - integral power of 2 725, 1506
 - inverse hyperbolic cosines 120
 - inverse hyperbolic sines 159
 - inverse hyperbolic tangents 167
 - logarithms 1539, 1541, 1543
 - modular arithmetic 675
 - next representable 1885
 - normalized fractions 725
 - not a number, determining if 1445
 - not safe to use in interrupt or signal handlers 50
 - powers 2129
 - printing 2133
 - radix-independent exponents 1545, 2705, 2707
 - remainders 2554
 - residue 675
 - rounding 260, 671, 2654, 2661

- floating point (*continued*)
 - settings 690, 693, 695, 697
 - exception mask 690
 - exception registers 693
 - precision 695
 - rounding 697
 - sign, copying 375
 - sines 3024
 - square roots 3153
 - tangents 3334
 - wide-character strings, converting to/from 3703
- flock 626
- flock() 666
- flockfile() 669
- floor(), floorf() 671
- flow control 3342
- flushall() 673
- fmod(), fmodf(), fmodl() 675
- FNM_PATHNAME 677
- FNM_PERIOD 677
- FNM_QUOTE 678
- fnmatch() 678
- fopen(), fopen64() 681
- fork() 685, 1913, 2143
- forkpty() 689
- fp_exception_mask() 690
- fp_exception_value() 693
- fp_precision() 695
- fp_rounding() 697
- fpathconf() 699
- fpos_t 650, 776
- fprintf() 702
- fputc() 704
- fputchar() 707
- fputs() 709
- fputwc() 711
- fputws() 713
- FQNN (Fully Qualified Node Name) 1878
- FQPN (Fully Qualified Path Name) 1877
- fractional part of a floating-point number 1689
- fread() 715
- free memory, amount of 3169
- free space, filesystem 784, 3171
- free() 717
- freeaddrinfo() 719
- freeifaddrs() 720
- freopen(), freopen64() 721
- frexp(), frexpf() 725
- fs_crypto_check() 727
- fs_crypto_domain_add() 729
- fs_crypto_domain_key_change() 732
- fs_crypto_domain_key_check() 735
- fs_crypto_domain_key_size() 738
- fs_crypto_domain_lock() 739
- fs_crypto_domain_query() 741
- fs_crypto_domain_remove() 743
- fs_crypto_domain_unlock() 745
- fs_crypto_enable_option() 750
- FS_CRYPT0_ENABLE_WHOLE_DISK 750
- fs_crypto_enable() 748
- fs_crypto_file_get_domain() 753
- fs_crypto_file_set_domain() 766
- fs_crypto_key_gen() 755
- fs_crypto_migrate_control() 757
- FS_CRYPT0_MIGRATE_DISABLE 757
- fs_crypto_migrate_path() 762
- FS_CRYPT0_MIGRATE_PAUSE 757
- FS_CRYPT0_MIGRATE_PERIOD 757
- FS_CRYPT0_MIGRATE_START 757
- fs_crypto_migrate_status() 760
- FS_CRYPT0_MIGRATE_STOP 757
- fs_crypto_migrate_tag() 764
- FS_CRYPT0_MIGRATE_UNITS 757
- FS_CRYPT0_REPLY_ALREADY 739, 746, 758, 767
- FS_CRYPT0_REPLY_BAD_FILE 767
- FS_CRYPT0_REPLY_BAD_KEY 733, 736, 746
- FS_CRYPT0_REPLY_BADF 762, 764
- FS_CRYPT0_REPLY_COMPLETE 730, 733, 736, 740, 743, 746, 748, 751, 753, 758, 761, 762, 765, 767
- FS_CRYPT0_REPLY_DOMAIN_EXISTS 730
- FS_CRYPT0_REPLY_ERRNO 730, 733, 736, 744, 746, 758, 767
- FS_CRYPT0_REPLY_INVALID 727, 730, 733, 736, 740, 741, 744, 746, 749, 751, 753, 758, 761, 763, 765, 767
- FS_CRYPT0_REPLY_LOCKED 741
- FS_CRYPT0_REPLY_NOENTRY 740, 741
- FS_CRYPT0_REPLY_NOSUPPORT 727, 758, 763, 765
- FS_CRYPT0_REPLY_NOT_ASSIGNED 754
- FS_CRYPT0_REPLY_NOT_ZERO 767
- FS_CRYPT0_REPLY_READONLY 730, 744, 749, 751, 763, 765, 767
- FS_CRYPT0_REPLY_SUPPORTED 727
- FS_CRYPT0_REPLY_UNKNOWN_DOMAIN 733, 736, 744, 746, 751, 767
- FS_CRYPT0_REPLY_UNKNOWN_TYPE 730, 736, 740, 746
- FS_CRYPT0_REPLY_UNLOCKED 742
- fs_crypto_set_logging() 769
- FS_CRYPT0_TYPE_CBC 729
- FS_CRYPT0_TYPE_NONE 729
- FS_CRYPT0_TYPE_XTS 729
- fscanf() 770
- fseek() 2646
- fseek(), fseeko(), fseeko64() 772
- fsetpos() 776
- fstab structure 844
- fstab table 495, 842, 844, 847, 2860
 - closing 495
 - getting next entry 844
 - opening 2860
 - searching for a block special device 847
 - searching for a filesystem 842
- FSTAB_IMPLIED 845
- FSTAB_OCB 845
- FSTAB_RO 845
- FSTAB_RW 845
- FSTAB_XX 845
- fstat(), fstat64() 779
- fstatvfs(), fstatvfs64() 783
- fsync() 787
- ftell(), ftello(), ftello64() 789
- ftime() 791
- ftruncate(), ftruncate64() 793
- ftrylockfile() 796
- FTW_D 799, 1889

FTW_DNR 799, 1889
 FTW_F 799, 1889
 FTW_NS 799, 1890
 ftw(), ftw64() 798
 full-duplex connection, shutting down 2942
 Fully Qualified Node Name (FQNN) 1878
 Fully Qualified Path Name (FQPN) 1877
 function 46, 49
 classification 46
 safety 49
 funlockfile() 801
 futime() 802
 fwide() 804
 fwprintf() 806
 fwrite() 808
 fwscanf() 810

G

gai_strerror() 814
 gamma functions 816, 1513
 gamma(), gamma_r(), gammaf(), gammaf_r() 816
 General Purpose Interface Bus (GPIB) 1949
 get_device_command() 406
 get_device_direction() 406
 getaddrinfo() 818
 getc_unlocked() 825
 getc() 823
 getchar_unlocked() 829
 getchar() 827
 getcwd() 830
 getdomainname() 833
 getdtablesize() 835
 getegid() 836
 getenv() 570, 592, 838, 3101, 3113, 3130, 3142
 geteuid() 840
 getfsent() 842, 844, 847
 getgid() 849
 getgrent() 851
 getgrgid_r() 855
 getgrgid() 853
 getgrnam_r() 859
 getgrnam() 857
 getgrouplist() 861
 getgroups() 864
 gethostbyaddr_r() 870
 gethostbyaddr() 866
 gethostbyname_r() 876
 gethostbyname(), gethostbyname2() 872
 gethostent_r() 881
 gethostent() 878
 gethostname() 883
 getifaddrs() 885
 GETIOVBASE() 887
 GETIOVLEN() 888
 getitimer() 889
 getlogin_r() 892
 getlogin() 891
 getnameinfo() 894
 getnetbyaddr() 898
 getnetbyname() 900
 getnetent() 902

getopt() 903
 getpagesize() 908
 getpagesizes(), getpagesizes64() 909
 getpass() 911
 getpeereid() 913
 getpeername() 915
 getpgid() 917, 2143
 getpgrp() 919
 getpid() 920
 getppid() 921
 getprio() 922
 getprotobyname() 924
 getprotobynumber() 926
 getprotoent() 928
 getpwent_r() 932
 getpwent() 930
 getpwnam_r() 937
 getpwnam() 934
 getpwuid_r() 941
 getpwuid() 938
 getrlimit(), getrlimit64() 943
 getrusage() 945
 gets() 950
 getservbyname() 952
 getservbyport() 954
 getservent() 956
 getsid() 958, 2143
 getsockname() 960, 3766
 getsockopt() 963
 getspent() 971
 getspnam(), getspnam_r() 975
 getsubopt() 977
 gettid() 981
 gettimeofday() 982
 getuid() 984
 getutent() 985
 getutid() 987
 getutline() 989
 getw() 991
 getwc() 993
 getwchar() 995
 getwd() 997
 glob_t 999
 glob() 1000
 global variables 151, 153, 154, 199, 237, 396, 491, 494,
 539, 540, 549, 903, 904, 2183, 2978, 3175,
 3176, 3177, 3329, 3475, 3546
 __progname 2183
 _ambblksiz 151
 _argc 153
 _argv 154
 _auxv 199
 _btext 237
 _edata 491
 _end 494
 _etext 549
 _syspage_ptr 3329
 daylight 396
 errno 539
 optarg 904
 opterr 904
 optind 903

global variables (*continued*)

- optopt 904
- stderr 3175
- stdin 3176
- stdout 3177
- sys_errlist 539
- sys_nerr 540
- sys_nsig 2978
- sys_siglist 2978
- timezone 3475
- tzname 3546
- globfree() 1004
- gmtime_r() 1007
- gmtime() 1005
- GPIO (General Purpose Interface Bus) 1949
- grantpt() 1009
- greater of two numbers 1582
- groups 496, 836, 849, 851, 853, 855, 857, 859, 861, 864, 917, 919, 1131, 2039, 2041, 2042, 2043, 2066, 2095, 2851, 2858, 2862, 2864, 2895, 3309
 - See also process groups
 - access list 861, 1131
 - getting 861
 - initializing 1131
 - database 496, 851, 1131, 2862
 - closing 496
 - membership 1131
 - next entry, getting 851
 - rewinding 2862
 - IDs 836, 849, 853, 855, 919, 2039, 2041, 2042, 2043, 2066, 2095, 2851, 2858, 2895
 - effective 836, 2039, 2041, 2042, 2043, 2851, 2858, 2895
 - information about 853, 855
 - process 919
 - real 849, 2039, 2041, 2042, 2043, 2066, 2095, 2858, 2895
 - saved 2858
 - name, getting information about 857, 859
 - set-group ID 2041, 3309
 - supplementary 864, 2864, 3309
 - IDs 864, 2864
 - maximum per process 3309
 - See also process groups
- guard area 908, 2193, 2217, 3309
- guardian process, specifying 2172

H

- h_errno 867, 873, 1012, 1017, 2570, 2576
- HAM (High Availability Manager) 394, 2157
- hardware 345, 1028, 1031, 1032, 1035, 1037, 1039, 3571
 - information in system page 1028, 1031, 1032, 1035, 1037, 1039
 - manufacturer, getting 345
 - serial number, getting 345
 - type 345, 3571
- hardware interrupts, See interrupts
- hash table 1014, 1016, 1021
 - creating 1014
 - destroying 1016
 - searching 1021

- hcreate() 1014
- hdestroy() 1016
- heap, allocating 151
- herror() 1017
- hexadecimal numbers 173, 1479, 1481
 - digit, testing a character for 1479, 1481
 - strings, converting to/from 173
- High Availability Manager (HAM) 394, 2157
- holding a process for debugging 2041, 2043, 3094
- HOST_NOT_FOUND 1012, 1017
- host-byte order 1026, 1027, 1896, 1897
 - network-byte order, converting to/from 1026, 1027, 1896, 1897
- HOSTALIASES 873
- hostent 1019
- HOSTNAME 883, 2868
- hosts 345, 497, 833, 866, 870, 872, 876, 878, 881, 883, 1012, 1017, 1019, 1024, 1094, 1096, 1419, 2696, 2849, 2866, 2868
 - addresses 1094, 1096
 - strings, converting to/from 1094, 1096
 - database 497, 866, 870, 872, 876, 878, 881, 1012, 1017, 1019, 1024, 2866
 - closing 497
 - entries, getting 866, 870, 872, 876, 878, 881
 - errors 1012, 1017, 1024
 - opening 2866
 - structure 1019
 - domain name 345, 833, 2849
 - getting 345, 833
 - setting 2849
 - names 345, 883, 2868
 - getting 345
 - valid characters 345
 - remote 1419, 2696
 - identity, checking 1419, 2696
- hot swapping 1997
- hsearch() 1021
- hstrerror() 1024
- htonl() 1026
- htons() 1027
- HUPCL 3381
- hwi_find_item() 1028
- hwi_find_tag() 1031
- hwi_next_item() 1032
- hwi_next_tag() 1035
- HWI_NULL_OFF 1029, 1032
- hwi_off2tag() 1037
- hwi_tag2off() 1039
- hyperbolic functions 120, 159, 167, 379, 3026, 3336
 - See also trigonometry
 - hyperbolic cosine 379
 - hyperbolic sine 3026
 - hyperbolic tangent 3336
 - inverse hyperbolic cosine 120
 - inverse hyperbolic sine 159
 - inverse hyperbolic tangent 167
 - See also trigonometry
- hypot(), hypotf() 1040
- hypotenuse, length of 1040

I

- I/O 247, 254, 267, 336, 381, 483, 485, 529, 623, 624, 625, 666, 678, 699, 700, 779, 783, 1148, 1155, 1162, 1172, 1174, 1176, 1179, 1181, 1516, 1519, 1535, 1639, 1642, 1905, 1916, 1936, 1937, 1943, 1945, 2040, 2048, 2050, 2052, 2131, 2156, 2470, 2502, 2506, 2520, 2523, 2673, 2674, 2792, 2979, 3084, 3088, 3170, 3276, 3308, 3345, 3409, 3410, 3518, 3519, 3558, 3737, 3742, 3745
 - asynchronous 699, 1936, 3308
 - priority for (`_SC_AIO_PRIO_DELTA_MAX`) 3308
 - support for (`_PC_ASYNC_IO`) 699, 1936
 - buffers, flushing 3345
 - configuration files, opening 267
 - end-of-file, checking for 529
 - FIFOs, creating 1639, 1642
 - file descriptors 483, 485, 624, 1916, 2040, 2050, 2673, 2792, 3088
 - duplicating 483, 485, 624, 1916, 2040, 2050, 3088
 - selecting 2673, 2792
 - file-mode creation mask 2156, 3558
 - daemons 2156
 - files 336, 381, 623, 624, 625, 666, 678, 779, 1516, 1535, 1905, 1943, 1945, 2040, 2048, 2052, 2131, 2470, 2502, 2506, 2520, 3084, 3276, 3737, 3742
 - closing 336, 2040, 2048, 2052
 - controlling 623
 - extending 623
 - information, getting 779
 - linking to 1516, 1943, 1945, 2520, 3276
 - locking 624, 1535
 - names, matching 678
 - opening 381, 1905, 2040, 3084
 - reading 2131, 2502, 2506
 - status flags 624
 - truncating 624
 - unlocking 625, 666, 1535
 - writing 2470, 3737, 3742
 - filesystems 783, 3170
 - information, getting 783, 3170
 - iov_t 2523, 3745
 - reading 2523
 - writing 3745
 - ports, managing 2674
 - prioritized, support for (`_PC_ASYNC_IO`) 700, 1937
 - privileges, requesting 247, 254, 1148, 1155, 1162, 1172, 1174, 1176, 1179, 1181, 2979, 3409, 3410, 3518, 3519
 - requests, initiating list of 1519
 - synchronous, support for (`_PC_ASYNC_IO`) 700, 1937
- I/O functions (resource managers) 1206, 1234, 1237, 1239, 1242, 1249, 1251, 1257, 1261, 1264, 1267, 1268, 1271, 1274, 1281, 1286, 1289, 1292, 1294, 1298, 1327, 1330, 1332, 1335, 1337, 1340, 1343, 1351, 1353, 1355, 1360, 1362, 1375, 1378, 1380
 - chmod 1234, 1237
 - default 1237
 - I/O functions (resource managers) (*continued*)
 - chown 1239, 1242
 - default 1242
 - close 1249, 1251
 - default 1249, 1251
 - default values, setting 1271
 - devctl 1206, 1257, 1261, 1264
 - default 1206, 1261
 - verification 1264
 - fdinfo 1267, 1268
 - default 1268
 - link 1274
 - lock 1281
 - default 1281
 - lseek 1286, 1289
 - default 1289
 - mknod 1292
 - mmap 1294, 1298
 - default 1298
 - openfd 1327, 1330
 - default 1330
 - pathconf 1332, 1335
 - default 1335
 - read 1337, 1340
 - default 1337
 - readlink 1343
 - space 1351
 - stat 1353, 1355
 - default 1355
 - sync 1360, 1362
 - default 1360
 - utime 1375, 1378
 - default 1378
 - write 1380
 - default 1380
 - I/O vector 887, 888, 2523, 2870, 3745
 - base, getting 887
 - fields, filling 2870
 - length, getting 888
 - reading from a file 2523
 - writing to a file 3745
 - ICANON 3381
 - ICMP (Internet Control Message Protocol) 1045
 - ICMP6 (Internet Control Message Protocol v6) 1047
 - ICMP6_FILTER 1047
 - ICMP6_FILTER_SETBLOCK() 1048
 - ICMP6_FILTER_SETBLOCKALL() 1048
 - ICMP6_FILTER_SETPASS() 1048
 - ICMP6_FILTER_SETPASSALL() 1048
 - ICMP6_FILTER_WILLBLOCK() 1048
 - ICMP6_FILTER_WILLPASS() 1048
 - ICRNL 3379
 - IEXTEN 3382
 - if_freenameindex() 1049
 - if_indextoname() 1050
 - if_msghdr 2665
 - if_nameindex 1052
 - if_nameindex() 1052
 - if_nametoindex() 1054
 - ifa_msghdr 2665
 - ifaddrs 1055
 - IFNAMSIZ 1050

- IGNBRK 3379
- IGNCR 3379
- IGNPAR 3379
- IHFLOW 3381
- ilogb(), ilogbf() 1057
- IN_ 1135, 1137
- IN_ACCESS 1135, 1137
- IN_ALL_EVENTS 1136
- IN_ATTRIB 1135, 1137
- IN_CLOSE 1135
- IN_CLOSE_NOWRITE 1135, 1137
- IN_CLOSE_WRITE 1135, 1137
- IN_CREATE 1135, 1137
- IN_DELETE 1135, 1137
- IN_DELETE_SELF 1135, 1137
- IN_DONT_FOLLOW 1136
- IN_EXCL_UNLINK 1136
- IN_IGNORED 1138
- IN_ISDIR 1138
- IN_MASK_ADD 1136
- IN_MODIFY 1135, 1137
- IN_MOVE 1135
- IN_MOVE_SELF 1135, 1138
- IN_MOVED_FROM 1135, 1137
- IN_ONESHOT 1136
- IN_ONLYDIR 1136
- IN_OPEN 1135, 1137
- IN_Q_OVERFLOW 1138
- IN_UNMOUNT 1138
- in16() 1063
- in16s() 1065
- in32() 1067
- in32s() 1069
- in6_pktinfo 1415
- in8() 1059
- in8s() 1061
- INADDR_ANY 1392, 3359
- INADDR_NONE 1073, 1074, 1087
- inbe16() 1063
- inbe32() 1067
- INCLUDE 2785
- incoming connections, listening for 1523, 2656
- index() 1071
- inet_addr() 1073
- INET_ADDRSTRLEN 1093
- inet_aton() 1075
- inet_pton() 1077
- inet_makeaddr() 1078
- inet_net_ntop() 1080
- inet_net_pton() 1085
- inet_netof() 1083
- inet_network() 1087
- inet_ntoa_r() 1091
- inet_ntoa() 1089
- inet_ntop() 1094
- inet_pton() 1096
- INET6 (Internet protocol v6 family) 1100
- INET6_ADDRSTRLEN 1093
- inet6_option_alloc() 1103, 1105, 1107, 1109, 1111, 1113
- inet6_option_append() 1103, 1105, 1107, 1109, 1111, 1113
- inet6_option_find() 1103, 1105, 1107, 1109, 1111, 1113
- inet6_option_init() 1103, 1105, 1107, 1109, 1111, 1113
- inet6_option_next() 1103, 1105, 1107, 1109, 1111, 1113
- inet6_option_space() 1103, 1105, 1107, 1109, 1111, 1113
- inet6_rthdr_add() 1115
- inet6_rthdr_getaddr() 1117
- inet6_rthdr_getflags() 1119
- inet6_rthdr_init() 1121
- inet6_rthdr_lasthop() 1123
- inet6_rthdr_reverse() 1125
- inet6_rthdr_segments() 1127
- inet6_rthdr_space() 1129
- infinite number, determining if 1441
- inherit mask 2041, 2043, 2076, 2105, 3094, 3095, 3413
- inheritance 3093
- INIT_PROCESS 987, 3595
- initgroups() 1131
- initstate() 1133
- INLCR 3379
- inle16() 1063
- inle32() 1067
- inodes 784, 2603, 3171
 - getting 2603
 - number of 784, 3171
- inotify_add_watch() 1136
- inotify_event 1137
- inotify_init() 1139
- inotify_rm_watch() 1141
- INPCK 3379
- input_line() 1143
- input, formatted 770, 810, 2713, 3618, 3622, 3626, 3649, 3748
- insque() 1145
- instruction set architecture 345
- instrumented kernel 1171
- integers 51, 56, 179, 181, 191, 193, 260, 447, 671, 2484, 2486, 2661, 3066
 - absolute value 56
 - atomic operations 179, 181, 191, 193
 - addition 179, 181
 - subtraction 191, 193
 - division 447
 - pseudo-random numbers 2484, 2486
 - quotient 447
 - remainder 447
 - rounding 260, 671, 2661
 - size of 51
 - system message log, writing to 3066
- integral logarithms 1057
- integral part of a floating-point number 1689
- integral power of 2 725, 1506
- Intel 80x86-specific interrupts 1185
- interfaces 1049, 1050, 1052, 1054
 - index, mapping to name 1050
 - list of, freeing 1049
 - list of, getting 1052
 - name, mapping to index 1054
- international characters, See wide characters
- Internet Control Message Protocol, See ICMP
- Internet domain 465, 466, 1012, 1017, 1024, 2563, 2567, 2570, 2573, 2576, 2578
 - errors 1012, 1017, 1024
 - name servers 2563, 2567, 2570, 2573, 2576, 2578
 - initializing 2563

Internet domain (*continued*)
 name servers (*continued*)
 querying 2567, 2570, 2573, 2576, 2578
 names 465, 466
 compressing 465
 expanding 466
 Internet Protocol, See IP
 InterruptAttach() 2144
 InterruptAttach(), InterruptAttach_r() 1148
 InterruptAttachEvent() 2144
 InterruptAttachEvent(), InterruptAttachEvent_r() 1154
 InterruptCharacteristic(), InterruptCharacteristic_r() 1159
 InterruptDetach(), InterruptDetach_r() 1162
 InterruptDisable() 1164
 InterruptEnable() 1166
 InterruptHookIdle() 1168
 InterruptHookTrace() 1171
 InterruptLock() 1174
 InterruptMask() 1176
 interrupts 50, 1148, 1150, 1154, 1159, 1162, 1164, 1166,
 1168, 1174, 1176, 1177, 1178, 1179, 1181,
 1183, 1185, 1983, 1985, 2674
 classes 1148
 disabling 1164
 enabling 1166
 events 1154, 1162
 attaching 1154
 detaching 1162
 handlers 50, 1148, 1150, 1162, 1164, 1168, 1174,
 1179
 attaching 1148
 detaching 1162
 disabling hardware interrupts 1164
 guidelines for writing 1150
 idle, attaching 1168
 locking 1174, 1179
 not safe to use floating-point in ISRs 50
 stack size 1150
 Intel 80x86-specific 1185
 level-sensitive 1177
 masking 1176, 1181
 PCI 1983, 1985
 mapping 1985
 routing information 1983
 requests, managing 2674
 status of 1159, 1178
 waiting for 1183
 InterruptStatus() 1178
 InterruptUnlock() 1179
 InterruptUnmask() 1181
 InterruptWait(), InterruptWait_r() 1183
 inverse hyperbolic cosines 120
 inverse hyperbolic sines 159
 inverse hyperbolic tangents 167
 io_acl_t 1213
 io_chmod_t 1235
 io_chown_t 1240
 io_close_t 1250
 io_devctl_t 1258
 io_fdinfo_t 1269
 io_link_t 1274
 io_lseek_t 1287
 io_mknod_t 1292
 io_mmap_t 1295
 io_notify_t 1302
 io_open_t 1323
 io_openfd_t 1328
 io_pathconf_t 1333
 io_read_t 1340
 io_readlink_t 1344
 io_rename_t 1347
 io_space_t 1351
 io_stat_t 1356
 io_sync_t 1362
 io_unlink_t 1371
 io_utime_t 1376
 io_write_t 1383
 ioctl_socket(), using instead of ioctl() 1198, 1199
 ioctl() 1198
 iofdinfo() 1211
 iofunc_acl_default() 1215
 iofunc_acl() 1212
 IOFUNC_ATTR_ETIME 1222, 1364, 1379
 IOFUNC_ATTR_CTIME 1222, 1364, 1379
 IOFUNC_ATTR_DIRTY_ACL 1222
 IOFUNC_ATTR_DIRTY_MODE 1222, 1379
 IOFUNC_ATTR_DIRTY_MTIME 1223
 IOFUNC_ATTR_DIRTY_NLINK 1223
 IOFUNC_ATTR_DIRTY_OWNER 1223
 IOFUNC_ATTR_DIRTY_RDEV 1223
 IOFUNC_ATTR_DIRTY_SIZE 1223
 IOFUNC_ATTR_DIRTY_TIME 1223, 1364, 1379
 iofunc_attr_init() 1218
 iofunc_attr_lock() 1219
 IOFUNC_ATTR_MTIME 1364, 1379
 IOFUNC_ATTR_PRIVATE 1223
 iofunc_attr_t 1222
 IOFUNC_ATTR_T 1320
 iofunc_attr_trylock() 1226
 iofunc_attr_unlock() 1228
 iofunc_check_access() 1231
 iofunc_chmod_default() 1237
 iofunc_chmod() 1234
 iofunc_chown_default() 1242
 iofunc_chown() 1239
 iofunc_client_info_able() 1245
 iofunc_client_info_ext_free() 1248
 iofunc_client_info_ext() 1247
 iofunc_close_dup_default() 1251
 iofunc_close_dup() 1249
 iofunc_close_ocb_default() 1255
 iofunc_close_ocb() 1253
 iofunc_devctl_default() 1206, 1261
 iofunc_devctl_verify() 1264
 iofunc_devctl() 1257
 iofunc_fdinfo_default() 1268
 iofunc_fdinfo() 1267
 iofunc_func_init() 1271
 iofunc_link() 1274
 iofunc_lock_calloc() 1279
 iofunc_lock_default() 1281
 iofunc_lock_free() 1283
 iofunc_lock_ocb_default() 1284
 iofunc_lock() 1276

- iofunc_lseek_default() 1289
- iofunc_lseek() 1286
- iofunc_mknod() 1292
- iofunc_mmap_default() 1298
- iofunc_mmap() 1294
- IOFUNC_MOUNT_32BIT 1287
- IOFUNC_MOUNT_T 1222
- IOFUNC_NOTIFY_DISARM() 1304
- IOFUNC_NOTIFY_INIT() 1304
- IOFUNC_NOTIFY_INPUT 1308
- IOFUNC_NOTIFY_INPUT_CHECK() 1309
- IOFUNC_NOTIFY_OBAND 1308
- IOFUNC_NOTIFY_OBAND_CHECK() 1309
- IOFUNC_NOTIFY_OUTPUT 1308
- IOFUNC_NOTIFY_OUTPUT_CHECK() 1309
- iofunc_notify_remove() 1306
- iofunc_notify_t 1303
- iofunc_notify_trigger() 1308
- iofunc_notify() 1302
- iofunc_ocb_attach() 1312
- iofunc_ocb_calloc() 1313
- iofunc_ocb_detach() 1315
- IOFUNC_OCB_FLAGS_PRIVATE 1320
- iofunc_ocb_free() 1317
- IOFUNC_OCB_PRIVILEGED 1320
- iofunc_ocb_t 1319
- iofunc_open_default() 1325
- iofunc_open() 1322
- iofunc_openfd_default() 1330
- iofunc_openfd() 1327
- iofunc_pathconf_default() 1335
- iofunc_pathconf() 1332
- IOFUNC_PC_ACL 1213, 1216
- iofunc_read_default() 1337
- iofunc_read_verify() 1340
- iofunc_readlink() 1343
- iofunc_rename() 1347
- iofunc_space_verify() 1351
- iofunc_stat_default() 1355
- iofunc_stat() 1353
- iofunc_sync_default() 1360
- iofunc_sync_verify() 1362
- iofunc_sync() 1358
- iofunc_time_update() 1364
- iofunc_unblock_default() 1368
- iofunc_unblock() 1366
- iofunc_unlink() 1371
- iofunc_unlock_ocb_default() 1373
- iofunc_utime_default() 1378
- iofunc_utime() 1375
- iofunc_write_default() 1380
- iofunc_write_verify() 1383
- ionotify() 1386
- iov_t 887, 888, 2523, 2632, 2870, 3745
 - base, getting 887
 - fields, filling 2870
 - length, getting 888
 - reading from a file 2523
 - resource managers 2632
 - writing to a file 3745
- iovec 1614
 - copying 1614
- IP (Internet Protocol) 1391
- IP addresses 1073, 1075, 1077, 1078, 1080, 1081, 1083, 1085, 1087, 1089, 1091, 1094, 1096
 - CIDR (Classless Internet Domain Routing), converting to/from 1080, 1085
 - classes of 1081
 - extracting 1083
 - network number 1083
 - local network addresses, extracting 1077
 - manipulating 1078
 - network numbers, converting to/from strings 1087
 - specifying in dot notation 1081
 - strings, converting to/from 1073, 1075, 1089, 1091, 1094, 1096
- IP Payload Compression Protocol 1400
- IP_ADD_MEMBERSHIP 1394
- IP_DROP_MEMBERSHIP 1394
- IP_HDRINCL 964, 1392
- IP_MAX_MEMBERSHIPS 1394
- ip_mreq 1394
- IP_MULTICAST_IF 1393
- IP_MULTICAST_LOOP 1393
- IP_MULTICAST_TTL 1392
- IP_OPTIONS 1391
- IP_RECVDSTADDR 1391
- IP_RECVIF 1392
- IP_TOS 964, 1391
- IP_TTL 1391
- IP6 (Internet Protocol v6) 1412
- IPPROTO_ICMPV6 1047
- IPPROTO_RAW 1392, 1417
- IPsec (secure Internet Protocol) 1396, 1402, 1404, 1406, 1410
 - errors 1410
 - policies 1402, 1404, 1406
- ipsec_dump_policy() 1402
- ipsec_get_policylen() 1404
- ipsec_set_policy() 1406
- ipsec_strerror() 1410
- IPv6 1103, 1105, 1107, 1109, 1111, 1113, 1121, 1129
 - hop-by-hop and destination options 1103, 1105, 1107, 1109, 1111, 1113
 - router header options 1121, 1129
- IPV6_BINDV6ONLY 1414
- IPV6_DSTOPTS 1414
- IPV6_HOPLIMIT 1414
- IPV6_HOPOPTS 1414
- IPV6_JOIN_GROUP 1413
- ipv6_mreq 1413
- IPV6_MULTICAST_HOPS 1412
- IPV6_MULTICAST_IF 1413
- IPV6_MULTICAST_LOOP 1413
- IPV6_PKTINFO 1414
- IPV6_PORTRANGE 1414
- IPV6_RTHDR 1414
- IPV6_UNICAST_HOPS 1412
- IRQs (Interrupt ReQuests), managing 2674
- iruserok(), iruserok_sa() 1419
- isalnum() 1421
- isalpha() 1423
- isascii() 1425
- isatty() 1427

isblank() 1429
 iscntrl() 1431
 isdigit() 1433
 isfdtype() 1435
 isfinite() 1437
 isgraph() 1439
 ISIG 3382
 isinf() 1441
 islower() 1443
 isnan() 1445
 isprint() 1447
 ispunct() 1449
 isspace() 1451
 ISTRIP 3380
 isupper() 1453
 iswalnum() 1455
 iswalph() 1457
 iswblank() 1459
 iswcntrl() 1461
 iswctype() 1463
 iswdigit() 1465
 iswgraph() 1467
 iswlower() 1469
 iswprint() 1471
 iswpunct() 1473
 iswspace() 1475
 iswupper() 1477
 iswxdigit() 1479
 isxdigit() 1481
 items 1145, 2559
 inserting into a list 1145
 removing from a list 2559
 ITIMER_REAL 890, 2873
 itimerspec 3437
 itoa() 1483
 IXOFF 3380
 IXON 3380

J

j0(), j0f() 1486
 j1(), j1f() 1488
 jn(), jnf() 1490
 job control, supporting 3309
 rand48() 1492
 jumps, nonlocal 1549, 2875, 2970, 3010

K

kernel 247, 254, 278, 281, 284, 304, 323, 325, 327, 329,
 333, 352, 356, 361, 363, 367, 370, 372, 397,
 398, 399, 1148, 1154, 1155, 1160, 1162, 1164,
 1166, 1168, 1171, 1174, 1176, 1179, 1181,
 1183, 1734, 1736, 1743, 1745, 1748, 1754,
 1757, 1760, 1766, 1769, 1771, 1776, 1779,
 1782, 1786, 1790, 1794, 1798, 1802, 1806,
 1810, 1814, 1817, 1820, 1823, 2745, 2774,
 2778, 2781, 2783, 2978, 2979, 2987, 2992,
 2995, 2997, 3280, 3282, 3286, 3289, 3291,
 3294, 3297, 3299, 3301, 3303, 3305, 3399,

kernel (*continued*)
 3403, 3408, 3417, 3420, 3422, 3449, 3452,
 3454, 3457, 3461, 3465, 3518, 3519
 blocking states, setting timeouts on 3465
 calls 247, 254, 278, 281, 284, 304, 323, 325, 327,
 329, 333, 352, 356, 361, 363, 367, 370,
 372, 397, 398, 399, 1148, 1154, 1155,
 1160, 1162, 1164, 1166, 1168, 1171, 1174,
 1176, 1179, 1181, 1183, 1734, 1736, 1743,
 1745, 1748, 1754, 1757, 1760, 1766, 1769,
 1771, 1776, 1779, 1782, 1786, 1790, 1794,
 1798, 1802, 1806, 1810, 1814, 1817, 1820,
 1823, 2745, 2774, 2778, 2781, 2783, 2978,
 2979, 2987, 2992, 2995, 2997, 3280, 3282,
 3286, 3289, 3291, 3294, 3297, 3299, 3301,
 3303, 3305, 3399, 3403, 3408, 3417, 3420,
 3422, 3449, 3452, 3454, 3457, 3461, 3465,
 3518, 3519
 ChannelCreate(), ChannelCreate_r() 278
 ChannelDestroy(), ChannelDestroy_r() 284
 ClientInfoExtFree() 304
 ClockAdjust(), ClockAdjust_r() 323
 ClockCycles() 325
 ClockId(), ClockId_r() 327
 ClockPeriod(), ClockPeriod_r() 329
 ClockTime(), ClockTime_r() 333
 ConnectAttach(), ConnectAttach_r() 352
 ConnectClientInfo(), ConnectClientInfo_r() 356
 ConnectClientInfoAble() 361
 ConnectClientInfoExt() 363
 ConnectDetach(), ConnectDetach_r() 281, 367
 ConnectFlags(), ConnectFlags_r() 370
 ConnectServerInfo(), ConnectServerInfo_r() 372
 DebugBreak() 397
 DebugKDBreak() 398
 DebugKDOutput() 399
 InterruptAttach(), InterruptAttach_r() 1148
 InterruptAttachEvent(), InterruptAttachEvent_r() 1154
 InterruptDetach(), InterruptDetach_r() 1162
 InterruptHookIdle() 1168
 InterruptHookTrace() 1171
 InterruptMask() 1176
 InterruptUnmask() 1181
 InterruptWait(), InterruptWait_r() 1183
 MsgCurrent(), MsgCurrent_r() 1734
 MsgDeliverEvent(), MsgDeliverEvent_r() 1736
 MsgError(), MsgError_r() 1743
 MsgInfo(), MsgInfo_r() 1745
 MsgKeyData(), MsgKeyData_r() 1748
 MsgRead(), MsgRead_r() 1754
 MsgReadv(), MsgReadv_r() 1757
 MsgReceive(), MsgReceive_r() 1760
 MsgReceivePulse(), MsgReceivePulse_r() 1766
 MsgReceivePulsev(), MsgReceivePulsev_r() 1769
 MsgReceivev(), MsgReceivev_r() 1771
 MsgReply(), MsgReply_r() 1776
 MsgReplyv(), MsgReplyv_r() 1779
 MsgSend(), MsgSend_r() 1782
 MsgSendnc(), MsgSendnc_r() 1786
 MsgSendPulse(), MsgSendPulse_r() 1790
 MsgSendsv(), MsgSendsv_r() 1794
 MsgSendsvnc(), MsgSendsvnc_r() 1798

kernel (*continued*)calls (*continued*)

MsgSendv(), MsgSendv_r() 1802
 MsgSendvnc(), MsgSendvnc_r() 1806
 MsgSendvs(), MsgSendvs_r() 1810
 MsgSendvsnc(), MsgSendvsnc_r() 1814
 MsgVerifyEvent(), MsgVerifyEvent_r() 1817
 MsgWrite(), MsgWrite_r() 1820
 MsgWritev(), MsgWritev_r() 1823
 SchedCtl(), SchedCtl_r() 2745
 SchedGet(), SchedGet_r() 2774
 SchedInfo(), SchedInfo_r() 2778
 SchedSet(), SchedSet_r() 2781
 SchedYield(), SchedYield_r() 2783
 SignalAction(), SignalAction_r() 2978
 SignalKill(), SignalKill_r() 2987
 SignalProcmask(), SignalProcmask_r() 2992
 SignalSuspend(), SignalSuspend_r() 2995
 SignalWaitinfo(), SignalWaitinfo_r() 2997
 SyncCondvarSignal(), SyncCondvarSignal_r() 3280
 SyncCondvarWait(), SyncCondvarWait_r() 3282
 SyncCtl(), SyncCtl_r() 3286
 SyncDestroy(), SyncDestroy_r() 3289
 SyncMutexEvent(), SyncMutexEvent_r() 3291
 SyncMutexLock(), SyncMutexLock_r() 3294
 SyncMutexRevive(), SyncMutexRevive_r() 3297
 SyncMutexUnlock(), SyncMutexUnlock_r() 3299
 SyncSemPost(), SyncSemPost_r() 3301
 SyncSemWait(), SyncSemWait_r() 3303
 SyncTypeCreate(), SyncTypeCreate_r() 3305
 ThreadCancel(), ThreadCancel_r() 3399
 ThreadCreate(), ThreadCreate_r() 3403
 ThreadCtl(), ThreadCtl_r() 247, 254, 1148, 1155, 1160, 1162, 1164, 1166, 1174, 1176, 1179, 1181, 2979, 3408, 3518, 3519
 ThreadDestroy(), ThreadDestroy_r() 3417
 ThreadDetach(), ThreadDetach_r() 3420
 ThreadJoin(), ThreadJoin_r() 3422
 TimerAlarm() 3449
 TimerCreate() 3452
 TimerDestroy(), TimerDestroy_r() 3454
 TimerInfo(), TimerInfo_r() 3457
 TimerSettime(), TimerSettime_r() 3461
 TimerTimeout(), TimerTimeout_r() 3465

debugging 398, 399

instrumented 1171

thread scheduler 2778

kill() 1494, 2148

killpg() 1496

L

L_ctermid 387

L_tmpnam 3480

labs() 1500

large-file support 46, 381, 779, 783, 793, 798, 943, 1556, 1558, 1600, 1656, 1905, 2030, 2131, 2470, 2899, 3164, 3170, 3373

directory hierarchy, walking 798

file information 779, 1558, 3164

filesystem information 783, 3170

mapped memory, offset of 1600, 2030

large-file support (*continued*)

opening 381, 1905

position 1556, 3373

setting 1556, 3373

reading 2131

shared memory, mapping 1656

symbolic link information 1558

system-resource limits 943, 2899

truncating 793

writing 2470

LC_ALL 2880

LC_COLLATE 2880

LC_CTYPE 1463, 2880

LC_MESSAGES 2880

LC_MONETARY 2880

LC_NUMERIC 2880

LC_TIME 2880

lchown() 1502

lcong48() 1505

lconv 1526

LD_BIND_NOW 459

LD_DEBUG 460

LD_DEBUG_OUTPUT 460

LD_LIBRARY_PATH 455

LD_PRELOAD 458, 460

ldexp(), ldexpf(), ldexpl() 1506

ldiv_t 1508

ldiv() 1508

length, calculating 1040, 3213, 3227, 3686

hypotenuse 1040

strings 3213, 3227

wide-character strings 3686

lesser of two numbers 1635

level-sensitive interrupts 1177

lfind() 1510

lgamma(), lgamma_r(), lgammaf(), lgammaf_r() 1513

LIB 2785

libraries 45, 345, 455, 460

linking against 45

loading before others 460

locating 345, 455

libsocks 3764

licenses 41

information about 41

limits 552, 700, 783, 835, 943, 1412, 1414, 1519, 1523, 1937, 2235, 2523, 2656, 2688, 2725, 2727, 2736, 2778, 2868, 2899, 2900, 3170, 3308, 3309, 3360

core files, size of 2900

data segment, size of 2900

device numbers 2688

files 700, 1937, 2900, 3309

descriptors, number of 2900

link count 700, 1937

maximum per process 3309

names, length of 700, 1937

filesystems 783, 3170

hops 1412, 1414

host names, length 2868

inheriting 552

iov arrays 2523

memory locked into physical memory 2900

limits (*continued*)

- path names, length of 700, 1937
- pipes, number of bytes written atomically 700, 1937
- processes 1519, 2725, 2727, 2736, 2900, 3308, 3309
 - argument lists 3308
 - CPU time 2900
 - execution time 2736
 - files, number open 3309
 - I/O requests 1519
 - mapped address space 2900
 - maximum per real user ID 2900, 3308
 - scheduling policy 2725, 2727
 - supplementary group IDs 3309
- resident set size 2900
- sockets, pending connections 1523, 2656
- stack size 2900
- system resources 835, 943, 2899
 - getting 835, 943
 - setting 2899
- TCP maximum segment size 3360
- terminals 700, 1937
 - canonical input buffer size 700, 1937
 - raw input buffer size 700, 1937
- threads 2235, 2778, 2900
 - execution time 2778
 - maximum number of 2900
 - priority 2778
 - stack size 2235
- total memory usage 2900
- line buffering, setting for stream I/O 2878
- linear search 1510, 1552
- LINES 300
- LINK_MAX 664, 1517
- link-local addresses 1101
- link() 1274, 1516
 - resource managers, implementing in 1274
- linker symbols 237, 491, 494, 549
 - _btext 237
 - _edata 491
 - _end 494
 - _etext 549
- links, symbolic 1502, 1558, 1943, 1945, 2520, 2531, 2556, 3276, 3579
 - creating 3276
 - deleting 2556, 3579
 - information, getting 1558
 - ownership, changing 1502
 - reading 2520
 - resolving 2531
 - temporary 1943, 1945
- lio_listio(), lio_listio64() 1519
- LIO_NOP 142, 1520
- LIO_NOWAIT 142, 1519
- LIO_READ 142, 1520
- LIO_WAIT 142, 1519
- LIO_WRITE 142, 1520
- listen() 1523, 3359, 3766
- lists 1145, 2559
 - inserting elements 1145
 - removing elements 2559
- little endian 51, 504, 506, 508, 510, 512, 514, 516, 518, 520, 1063, 1067, 1626, 1927, 1931, 3270, 3561, 3563, 3565, 3567, 3568, 3569
 - __LITTLEENDIAN__ manifest 51
 - big endian, converting to/from 510, 512, 514, 516, 518, 520, 3270
 - messages 1626
 - native format, converting to/from 504, 506, 508
 - ports 1063, 1067, 1927, 1931
 - reading from 1063, 1067
 - writing to 1927, 1931
 - unaligned values 3561, 3563, 3565, 3567, 3568, 3569
 - accessing safely 3567, 3568, 3569
 - writing safely 3561, 3563, 3565
- llabs() 1525
- lldiv_t 1508
- lldiv() 1508
- lltoa() 1560
- local network addresses, converting to/from IP addresses 1078
- local times, converting to/from calendar times 1531, 1533, 1649
- LOCAL_CREDS 3578
- LOCALDOMAIN 2563, 2568, 2571, 2573, 2576, 2579
- localeconv() 1526
- locales 345, 396, 1526, 2880, 3190, 3678, 3722
 - classes, wide-character 3722
 - current (_CS_LOCALE) 345
 - daylight saving time 396
 - numeric formatting 1526
 - setting 2880
 - strings, comparing 3190, 3678
- localtime_r() 1533
- localtime() 1531
- LOCK_EX 666
- LOCK_NB 666
- LOCK_SH 666
- LOCK_UN 666
- lockf(), lockf64() 1535
- locks 624, 666, 669, 796, 1535, 1651, 1653, 1828, 1830, 2312, 2314, 2316, 2318, 2322, 2324, 2327, 2329, 2333, 2335, 2337, 2339, 2341, 2343, 2347, 2349, 2352, 2353, 2355, 2358, 2364, 2366, 2369, 2371, 2373, 2375, 2377, 2379, 2381, 2383, 2385, 2387, 2389, 2409, 2411, 2413, 2415, 2418, 2419, 3030, 3032, 3034, 3036, 3038, 3040, 3042, 3286, 3289, 3291, 3294, 3297, 3299, 3305
 - files 624, 666, 669, 796, 1535
 - memory 1651, 1653, 1828, 1830
 - creating 1651, 1653
 - releasing 1828, 1830
- mutexes 2312, 2314, 2316, 2318, 2322, 2324, 2327, 2329, 2333, 2335, 2337, 2339, 2341, 2343, 2347, 2349, 2352, 2353, 2355, 2358, 3286, 3289, 3291, 3294, 3297, 3299, 3305
 - attributes, destroying 2333
 - attributes, initializing 2347
 - attributes, priority ceiling 2335, 2349
 - attributes, process-shared 2339, 2353
 - attributes, recursive 2341, 2355
 - attributes, scheduling protocol 2337, 2352
 - attributes, type 2343, 2358

- locks (*continued*)
 - mutexes (*continued*)
 - destroying 2312, 3289
 - events 3286, 3291
 - initializing 2316, 3305
 - locking 2318, 2324, 2327, 3294, 3299
 - priority 3286
 - priority ceiling 2314, 2322
 - reviving 3297
 - unlocking 2329
 - read-write 2364, 2366, 2369, 2371, 2373, 2375, 2377, 2379, 2381, 2383, 2385, 2387, 2389, 3289, 3305
 - attributes, creating 2387
 - attributes, destroying 2383
 - attributes, process-shared 2385, 2389
 - destroying 2364, 3289
 - initializing 2366, 3305
 - locking for reading 2369, 2371, 2375
 - locking for writing 2373, 2377, 2381
 - unlocking 2379
 - sleepon 2409, 2411, 2413, 2415, 2418, 2419, 3030, 3032, 3034, 3036, 3038, 3040, 3042
 - destroying 3032
 - initializing 3034
 - locking 2411, 3036
 - unblocking 2409, 2413, 3030, 3038
 - unlocking 2418, 3040
 - waiting 2415, 2419, 3042
 - log 2749
 - bankruptcy (not implemented) 2749
 - LOG_* (logging facilities) 1919
 - LOG_ALERT 3317
 - LOG_CRIT 3317
 - LOG_DEBUG 3317
 - LOG_EMERG 3317
 - LOG_ERR 3317
 - LOG_INFO 3317
 - LOG_MASK() 2882
 - LOG_NOTICE 3317
 - LOG_UPTO() 2882
 - LOG_WARNING 3317
 - log, system message 340, 1920, 2882, 3061, 3063, 3066, 3316, 3631, 3643
 - closing 340
 - log priority mask 2882
 - opening 1920
 - writing to 3061, 3063, 3066, 3316, 3631, 3643
 - blocks 3061
 - formatted output 3063, 3631
 - integers 3066
 - log, system message (slogger2) 3045, 3047, 3049, 3051, 3052, 3056, 3057, 3059, 3629
 - default buffer, setting 3057
 - logging a constant string 3045
 - logging a formatted string 3047, 3049, 3629
 - registering with 3052
 - resetting 3056
 - verbosity 3051, 3059
 - log(), logf(), logl() 1539
 - log10(), log10f(), log10l() 1541
 - log1p(), log1pf(), log1pl() 1543
 - logarithms 1057, 1539, 1541, 1543
 - base 10 1541
 - integral 1057
 - natural 1539
 - $x + 1$ 1543
 - logb(), logbf() 1545
 - logging in 689, 1547, 1677
 - previous lines, discarding 1677
 - pseudo-ttys 689, 1547
 - logical interrupt vector numbers 1148
 - LOGIN_PROCESS 987, 989, 3595
 - login_tty() 1547
 - long integers 51, 177, 1492, 1500, 1508, 1551, 1560, 1730, 1894, 2487, 3262, 3264, 3555, 3709, 3713
 - size of 51
 - absolute value 1500
 - converting from strings 177, 3262, 3264
 - converting from wide-character strings 3709, 3713
 - converting to strings 1560, 3555
 - division 1508
 - pseudo-random numbers 1492, 1551, 1730, 1894, 2487
 - nonnegative 1551, 1894
 - signed 1492, 1730, 2487
 - quotient 1508
 - remainder 1508
 - long long integers 177, 1508, 1525, 1560, 3262, 3264, 3555, 3709, 3713
 - absolute value 1525
 - converting from strings 177, 3262, 3264
 - converting from wide-character strings 3709, 3713
 - converting to strings 1560, 3555
 - quotient 1508
 - remainder 1508
 - longjmp() 1549, 2482
 - lowercase 1443, 1469, 3215, 3482, 3486, 3488, 3720
 - characters, converting to 3482
 - strings, converting to 3215
 - testing a character for 1443, 1469
 - wide characters, converting to 3486, 3488, 3720
 - lrand48() 1551
 - lsearch() 1552
 - lseek(), lseek64() 1286, 1289, 1556
 - resource managers, implementing in 1286, 1289
 - lstat(), lstat64() 1558
 - ltoa() 1560
 - ltrunc() 1563
- ## M
- M_HANDLE_ABORT 1580
 - M_HANDLE_CORE 1580
 - M_HANDLE_EXIT 1580
 - M_HANDLE_IGNORE 1580
 - M_HANDLE_SIGNAL 1580
 - main() 153, 154, 199, 903, 1566, 2974
 - arguments 153, 154, 199, 903
 - auxiliary 199
 - number of 153
 - parsing 903
 - vector of 154

major device numbers 2603, 2688, 3167
 major() 3168
 makedev() 3168
 mallinfo 1568
 mallinfo() 1568
 MALLOC_ 1578
 MALLOC_ARENA_CACHE_FREE_NOW 1577
 MALLOC_ARENA_CACHE_MAXBLK 1570, 1577
 MALLOC_ARENA_CACHE_MAXSZ 1570, 1577
 MALLOC_ARENA_SIZE 152, 1571, 1576
 MALLOC_BAND_CONFIG_STR 1572
 MALLOC_BTDEPTH 1573
 MALLOC_CKACCESS 1578
 MALLOC_CKACCESS_LEVEL 1573
 MALLOC_CKBOUNDS 1578
 MALLOC_CKCHAIN 1578
 MALLOC_DUMP_LEAKS 1573, 1579
 MALLOC_EVENTBTDEPTH 1579
 MALLOC_FATAL 1580
 MALLOC_FREE_CHECK 1577
 MALLOC_FREE_LIFO 1571
 MALLOC_INITVERBOSE 1573
 MALLOC_MAX_ALIGNMENT 1604, 2034
 MALLOC_MEMORY_HOLD 1571, 1577
 MALLOC_MEMORY_PREALLOCATE 1572
 MALLOC_MMAP_NOZERO 1572
 MALLOC_OPTIONS 255, 256, 1569, 1572, 2527, 2528, 2529
 MALLOC_START_TRACING 1579
 malloc_stats 1577
 MALLOC_STATS 1577
 MALLOC_TRACE 1573
 MALLOC_TRACEBT 1573
 MALLOC_TRACEBTDEPTH 1580
 MALLOC_TRACEMAX 1579
 MALLOC_TRACEMIN 1579
 MALLOC_TRACING 1580
 MALLOC_VERIFY 1580
 MALLOC_VERIFY_ON 1577
 MALLOC_WARN 1580
 malloc() 1569
 mallopt() 152, 1575
 manifests 51
 MAP_ANON 1659
 MAP_BELOW 1659
 MAP_BELOW16M 1660
 MAP_ELF 1660
 MAP_FAILED 1663, 1670
 MAP_FILE 1660
 MAP_FIXED 1660, 1669, 1670, 2144
 MAP_LAZY 1660, 2209, 2233
 preallocating for a thread's stack 2209, 2233
 MAP_NOINIT 1661, 1839
 MAP_NORESERVE 1661
 MAP_NOSYNCFILE 1661
 MAP_NOX64K 1661
 MAP_PHYS 1661, 2144
 MAP_PRIVATE 1659, 2181
 MAP_PRIVATEANON 1662
 MAP_RENAME 1661
 MAP_SHARED 1659, 1660
 MAP_STACK 1662
 MAP_SYSRAM 1662
 MAP_TYPE 1659
 mathematics 56, 118, 120, 157, 159, 163, 165, 167, 242, 258, 260, 375, 377, 379, 447, 468, 531, 532, 534, 601, 603, 606, 671, 675, 690, 693, 695, 697, 725, 816, 1040, 1057, 1133, 1437, 1441, 1445, 1486, 1488, 1490, 1492, 1500, 1505, 1506, 1508, 1513, 1525, 1539, 1541, 1543, 1545, 1551, 1582, 1635, 1730, 1885, 1894, 2129, 2484, 2486, 2487, 2554, 2654, 2661, 2705, 2707, 2787, 2913, 3024, 3026, 3153, 3155, 3157, 3158, 3334, 3336, 3754, 3756, 3758
 absolute values 56, 242, 606, 1500, 1525
 Bessel functions 1486, 1488, 1490, 3754, 3756, 3758
 complementary error function 534
 division 447, 1508
 error function 532
 exponentials 601, 603, 725, 1506, 2705, 2707
 finite numbers 1437
 floating-point settings 690, 693, 695, 697
 gamma functions 816, 1513
 hyperbolic functions 120, 159, 167, 379, 3026, 3336
 hypotenuse, length of 1040
 infinite numbers 1441
 logarithms 1057, 1539, 1541, 1543
 maximum 1582
 minimum 1635
 modular arithmetic 675
 next representable number 1885
 not a number, determining if 1445
 powers 2129
 pseudo-random numbers 468, 531, 1133, 1492, 1505, 1551, 1730, 1894, 2484, 2486, 2487, 2787, 2913, 3155, 3157, 3158
 radix-independent exponents 1545
 remainders 2554
 roots 258, 3153
 rounding 260, 671, 2654, 2661
 sign, copying 375
 trigonometry 118, 157, 163, 165, 377, 3024, 3334
 max() 1582
 MAXHOSTNAMELEN 2868
 Maximum Segment Size (MSS) 3360
 MB_CUR_MAX 3718
 mblen() 1583
 mbrlen() 1585
 mbrtowc() 1587
 mbsinit() 1589
 mbsrtowcs() 1591
 mbstate_t 1589
 mbstowcs() 1593
 mbtowc() 1595
 MCHECK_DISABLED 1698
 MCHECK_FREE 1698
 MCHECK_HEAD 1698
 MCHECK_OK 1698
 mcheck_status 1697
 MCHECK_TAIL 1698
 mcheck() 1598
 mem_offset(), mem_offset64() 1600
 memalign() 1603

- members, offset of within a structure 1900
- memcpy() 1605
- memchr() 1607
- memcmp() 1609
- memcpy() 1611
- memcpyv() 1614
- memicmp() 1616
- memmove() 1618
- memory 147, 151, 213, 255, 272, 717, 1568, 1569, 1575, 1598, 1600, 1603, 1605, 1607, 1609, 1611, 1614, 1616, 1618, 1620, 1622, 1651, 1653, 1656, 1661, 1666, 1669, 1697, 1699, 1825, 1828, 1830, 1832, 1834, 1836, 1838, 2028, 2030, 2033, 2122, 2124, 2209, 2233, 2527, 2674, 2702, 2709, 2900, 2923, 2925, 2931, 2935, 2940, 3068, 3160, 3169, 3309, 3409, 3410, 3610, 3724, 3726, 3728, 3730, 3732
 - advisory information 2028
 - allocating 147, 151, 213, 255, 1568, 1569, 1575, 1598, 1603, 1656, 1661, 1697, 2033, 2527, 2702, 2709, 3068, 3610
 - _amblksiz 3068
 - aligned 1603, 2033
 - array 255, 2709
 - automatic (from stack) 147
 - blocks 1569, 2527, 3068
 - break value, changing 151, 2702
 - consistency check 1598, 1697
 - contiguous 1661
 - controlling 1575
 - data segment, changing 213
 - debugging 1575
 - heap block, aligned on page boundary 3610
 - information about, getting 1568
- comparing 1609, 1616, 3726
- copying 1605, 1611, 1614, 1618, 3728, 3730
 - overlapping objects 1618, 3730
- devices 1666, 1669, 1834
 - I/O, mapping 1666, 1834
 - physical, mapping into process's address space 1669
- direct memory access (DMA) channels, managing 2674
- free, amount of 3169
- freeing 272, 717, 2527, 2923, 3160
- locking 1651, 1653, 2900
 - maximum amount 2900
- managing 2674
- mapped 1600, 2030, 2900
 - contiguous length 1600, 2030
 - maximum size 2900
 - offset of 1600, 2030
- mapping 1656
- physical storage, synchronizing with 1825
- preallocating for a thread's MAP_LAZY stack 2209, 2233
- reallocating 2527, 3160
- resource constrain threshold 3309
- searching 1607, 3724
 - for a character 1607
 - for a wide character 3724
- setting 1620, 1622, 3732
- shared 1656, 1699, 1832, 1838, 2925, 2931, 2935, 2940
 - access protection, changing 1699
- memory (*continued*)
 - shared (*continued*)
 - attributes, modifying 2925, 2931
 - mapping 1656
 - opening 2935
 - removing 2940
 - unmapping 1832, 1838
 - superlocking 3409, 3410
 - total usage, limiting 2900
 - typed 1656, 2122, 2124
 - mapping 1656
 - opening 2124
 - querying 2122
 - unlocking 1828, 1830
 - unmapping 1836
- memory swapper 2148
- memset_s() 1622
- memset() 1620
- message queues 1702, 1704, 1707, 1709, 1710, 1711, 1714, 1717, 1720, 1722, 1725, 1728
 - attributes 1704, 1720
 - closing 1702
 - messages 1714, 1717, 1722, 1725
 - receiving 1714, 1722
 - sending 1717, 1725
 - notifying when nonempty 1707
 - opening 1710
 - persistence of 1711
 - receive-only 1709
 - send-only 1709
 - send-receive 1709
 - unlinking 1728
- Message Signaled Interrupts (MSI) 1970
- message_attach() 1625
- message_attr_t 1625
- message_connect() 1630
- message_detach() 1633
- messages 278, 284, 352, 465, 466, 1012, 1017, 1024, 1625, 1633, 1731, 1734, 1743, 1745, 1748, 1754, 1757, 1760, 1771, 1776, 1779, 1782, 1786, 1794, 1798, 1802, 1806, 1810, 1814, 1820, 1823, 1842, 1850, 2533, 2534, 2536, 2537, 2539, 2540, 2563, 2567, 2570, 2573, 2576, 2578, 2589, 2608, 2618, 2620, 2622, 2625, 2628, 2630, 2830, 2834, 2838
 - channels 278, 284, 352, 1842, 1850
 - attaching to a process 352
 - creating 278, 1842
 - destroying 284, 1850
 - dispatch interface 1625, 1633
 - handlers 1625, 1633
 - errors, handling 1743
 - information about 1731, 1745
 - getting 1745
 - structure 1731
 - Internet domain name servers 465, 466, 1012, 1017, 1024, 2563, 2567, 2570, 2573, 2576, 2578
 - errors 1012, 1017, 1024
 - queries 2567, 2570, 2573, 2576, 2578
 - sending and interpreting 465, 466, 2563
 - key, adding 1748
 - reading data 1754, 1757, 2618, 2620

messages (*continued*)
 receiving 1760, 1771
 replying 1776, 1779, 2622, 2625
 resource managers 2589, 2608
 blocking while waiting for 2589
 handling 2608
 resume processing 1734
 sending 1782, 1786, 1794, 1798, 1802, 1806, 1810, 1814
 sockets 2533, 2534, 2536, 2537, 2539, 2540, 2830, 2834, 2838
 peeking at 2533, 2536, 2539
 receiving from 2534, 2537, 2540
 sending to 2830, 2834, 2838
 tampering, preventing 1748
 unblocking 1743
 writing data 1820, 1823, 2628, 2630
 min() 1635
 minor device numbers 2603, 2688, 3167
 minor() 3168
 misaligned access response 3409
 mkdir() 1636, 3558
 mkfifo() 1639, 3558
 mknod() 1223, 1292, 1642
 resource managers, implementing in 1292
 mkstemp() 1645
 mktemp() 1647
 mktime() 1649
 mlock() 1651, 2144
 mlockall() 1653, 2144
 mmap_device_io() 1666, 2144
 mmap_device_memory() 1669, 2144
 mmap() 151, 2144, 2146
 mmap(), mmap64() 1294, 1298, 1656
 resource managers, implementing in 1294, 1298
 MODEM_ALLOW8BIT 1677
 MODEM_ALLOWCASE 1676
 MODEM_ALLOWCTRL 1676
 MODEM_BAUD 1680, 1684
 MODEM_LASTLINE 1677
 MODEM_NOECHO 1680, 1681, 1684
 modem_open() 1672
 modem_read() 1677
 modem_script 1680
 modem_script() 1680
 modem_write() 1686
 modems 1672, 1677, 1680, 1682, 1686
 opening 1672
 reading 1677
 script, running on 1680, 1682
 states 1682
 writing 1686
 escape characters 1686
 special characters 1686
 modf(), modff() 1689
 modular arithmetic, floating point 675
 mount_parse_generic_args() 1694
 mount() 1692
 mprobe() 1697
 mprotect() 1699
 mq_attr 1704
 mq_close() 1702
 mq_getattr() 1704
 mq_notify() 1707
 mq_open() 1710
 MQ_PRIO_MAX 1717, 1725
 mq_receive() 1714
 mq_send() 1717
 mq_setattr() 1720
 mq_timedreceive(), mq_timedreceive_monotonic() 1722
 mq_timedsend(), mq_timedsend_monotonic() 1725
 mq_unlink() 1728
 mrand48() 1730
 MS_ASYNC 1825
 MS_CACHE_ONLY 1825
 MS_INVALIDATE 1825
 MS_INVALIDATE_ICACHE 1825
 MS_SYNC 1826
 MSG_CTRUNC 2540
 MSG_DONTROUTE 2829, 2833, 2837
 MSG_EOR 2540
 MSG_FLAG_CROSS_ENDIAN 441
 MSG_FLAG_SIDE_CHANNEL 1630
 MSG_NOSIGNAL 2829, 2833, 2837
 MSG_OOB 966, 2533, 2536, 2539, 2540, 2829, 2833, 2837
 MSG_PEEK 2533, 2536, 2539
 MSG_TRUNC 2540
 MSG_WAITALL 2533, 2536, 2539
 MsgCurrent(), MsgCurrent_r() 1734
 MsgDeliverEvent(), MsgDeliverEvent_r() 1388, 1736
 MsgError(), MsgError_r() 1743
 msghdr 2540
 MsgInfo(), MsgInfo_r() 1745
 MsgKeyData() 2144
 MsgKeyData(), MsgKeyData_r() 1748
 MsgRead(), MsgRead_r() 1754
 MsgReadv(), MsgReadv_r() 1757
 MsgReceive(), MsgReceive_r() 1760
 MsgReceivePulse(), MsgReceivePulse_r() 1766
 MsgReceivePulseev(), MsgReceivePulseev_r() 1769
 MsgReceivev(), MsgReceivev_r() 1771
 MsgReply(), MsgReply_r() 1776
 MsgReplyv(), MsgReplyv_r() 1779
 MsgSend(), MsgSend_r() 1782
 MsgSendnc(), MsgSendnc_r() 1786
 MsgSendPulse() 2143
 MsgSendPulse(), MsgSendPulse_r() 1790
 MsgSendsv(), MsgSendsv_r() 1794
 MsgSendsvnc(), MsgSendsvnc_r() 1798
 MsgSendv(), MsgSendv_r() 1802
 MsgSendvnc(), MsgSendvnc_r() 1806
 MsgSendvs(), MsgSendvs_r() 1810
 MsgSendvsnc(), MsgSendvsnc_r() 1814
 MsgVerifyEvent(), MsgVerifyEvent_r() 1817
 MsgWrite(), MsgWrite_r() 1820
 MsgWritev(), MsgWritev_r() 1823
 MSI (Message Signaled Interrupts) 1970
 MSI-X (Extended Message Signaled Interrupts) 1970
 MSS (Maximum Segment Size) 3360
 msync() 1825
 multibyte characters 1583, 1585, 1587, 1589, 1591, 1593, 1595, 3671, 3697, 3711, 3718
 bytes, counting 1583, 1585

- multibyte characters (*continued*)
 - wide characters 1589
 - conversion object 1589
 - wide characters, converting to/from 1587, 1591, 1593, 1595, 3671, 3697, 3711, 3718
- multicore systems 3517
 - CPU index in trace events 3517
- munlock() 1828
- munlockall() 1830
- munmap_device_io() 1834
- munmap_device_memory() 1836
- munmap_flags() 1838
- munmap() 1832
- mutexes 2312, 2314, 2316, 2318, 2322, 2324, 2327, 2329, 2331, 2333, 2335, 2337, 2339, 2341, 2343, 2345, 2347, 2349, 2352, 2353, 2355, 2358, 2359, 3286, 3289, 3291, 3294, 3297, 3299, 3305
 - attributes 2333, 2335, 2337, 2339, 2341, 2343, 2345, 2347, 2349, 2352, 2353, 2355, 2358, 2359
 - destroying 2333
 - initializing 2347
 - priority ceiling 2335, 2349
 - process-shared 2339, 2353
 - recursive 2341, 2355
 - scheduling protocol 2337, 2352
 - type 2343, 2358
 - wake-up 2345, 2359
 - destroying 2312, 3289
 - events, attaching 3286, 3291
 - initializing 2316, 3305
 - locking 2318, 2324, 2327, 3294, 3299
 - priority 2314, 2322, 3286
 - ceiling 2314, 2322
 - reviving 3297
 - unlocking 2329
 - waking up blocked threads 2331, 3286

N

- name servers 465, 466, 1012, 1017, 1024, 2563, 2567, 2570, 2573, 2576, 2578
 - errors 1012, 1017, 1024
 - initializing 2563
 - names 465, 466
 - compressing 465
 - expanding 466
 - queries 2567, 2570, 2573, 2576, 2578
- NAME_ATTACH_FLAG_GLOBAL 1843
- name_attach_t 1843
- name_attach() 436, 1842
- name_close() 1848
- name_detach() 1850
- NAME_FLAG_ATTACH_GLOBAL 1842, 1852
- NAME_FLAG_DETACH_SAVEDPP 1850
- NAME_MAX 1853
- name_open() 1852
- names 208, 345, 833, 883, 915, 960, 2297, 2398, 2489, 2650, 2849, 2868, 3410
 - domain 345, 833, 2849
 - host 883, 2868

- names (*continued*)
 - peer 915
 - socket 208, 960, 2489, 2650
 - thread 2297, 2398, 3410
- NAN (not-a-number) 1445
- nanoseconds 1855, 1857, 1859, 1861, 1863, 1865, 1895, 3474
 - busy-waiting for 1857, 1859, 1861, 1863, 1865
 - threads, suspending for 1855
 - timespec, converting to/from 1895, 3474
- nanosleep() 1855
- nanospin_calibrate() 1859
- nanospin_count() 1861
- nanospin_ns_to_count() 1865
- nanospin_ns() 1863
- nanospin() 1857
- nap() 1867
- napms() 1868
- natural logarithms 1539
- nbaconnect_result() 1871
- nbaconnect() 1869
- ncurses 1873
- ND_LOCAL_NODE 1874
- ND_NODE_CMP() 1874
- ND2S_DIR_HIDE 1878
- ND2S_DIR_SHOW 1878
- ND2S_DOMAIN_HIDE 1878
- ND2S_DOMAIN_SHOW 1878
- ND2S_LOCAL_STR 1878
- ND2S_NAME_HIDE 1879
- ND2S_NAME_SHOW 1879
- ND2S_QOS_HIDE 1879
- ND2S_QOS_SHOW 1879
- ND2S_SEP_FORCE 1879
- NDEBUG 161
- NEON, determining if supported 3326
- NetBSD classification 46
- NETDB_INTERNAL 873, 1012, 1017
- netent 1876
- netmgr_ndtostr() 1877
- netmgr_remote_nd() 1881
- netmgr_strtond() 1883
- network 522, 866, 870, 872, 876, 878, 881, 898, 900, 902, 1012, 1017, 1024, 1876, 2884
 - database 522, 898, 900, 902, 1876, 2884
 - closing 522
 - entries, getting 898, 900, 902
 - opening 2884
 - structure 1876
 - host entries 866, 870, 872, 876, 878, 881, 1012, 1017, 1024
 - errors 1012, 1017, 1024
 - getting 866, 870, 872, 876, 878, 881
- network interface addresses 720, 885, 1055
 - freeing 720
 - getting 885
 - structure 1055
- network numbers 1078, 1083, 1087
 - IP addresses, converting to/from 1078, 1083
 - strings, converting to/from 1087

network-byte order 1026, 1027, 1896, 1897
 host-byte order, converting to/from 1026, 1027, 1896, 1897
 Neutrino classification 48
 NEW_TIME 987, 3594
 nextafter(), nextafterf() 1885
 nftw(), nftw64() 1888
 NGROUPS_MAX 356
 nice() 1892
 NO_DATA 1012, 1018
 NO_RECOVERY 1013, 1018
 node descriptors 1874, 1877, 1881, 1883, 3571
 comparing 1874
 current 3571
 relative to a remote node 1881
 strings, converting to/from 1877, 1883
 nodenames, translating addresses into 894
 NOFD 1600, 1657, 1659
 NOFLSH 3382
 nonlocal jumps 1549, 2875, 2970, 3010
 normalized fractions 725
 NOSIOCGIFCONF 1199
 not a number, determining if 1445
 nrand48() 1894
 nsec2timespec() 1895
 NSIG 2963
 ntohl() 1896
 ntohs() 1897
 numbers 171, 173, 175, 177, 1437, 1441, 1445, 1483, 1526, 1560, 1582, 1635, 1885, 3253, 3257, 3262, 3264, 3555, 3598, 3703, 3706, 3709, 3713
 determining if 1437, 1441, 1445
 finite 1437
 infinite 1441
 not a number 1445
 formatting 1526
 maximum 1582
 minimum 1635
 next representable floating-point 1885
 strings, converting to/from 171, 173, 175, 177, 1483, 1560, 3253, 3257, 3262, 3264, 3555, 3598
 wide-character strings, converting to/from 3703, 3706, 3709, 3713

O

O_APPEND 138, 624, 1191, 1328, 1906, 1916, 3083, 3087, 3737
 O_APPEND (not currently supported) 1906
 O_ASYNC 1199
 O_CLOEXEC 1906
 O_CREAT 381, 1709, 1906, 2935, 3083, 3084
 O_DSYNC 129, 1191, 1358, 1363, 1907, 3743
 O_EXCL 1191, 1709, 1907, 1958, 2935, 3083
 O_LARGEFILE 1191, 1907
 O_NOCTTY 1191, 1907, 2035
 O_NOFOLLOW 1907
 O_NONBLOCK 624, 1199, 1339, 1350, 1382, 1678, 1709, 1907, 2011, 2503, 2513, 2524, 3738
 O_NOSYMLINK 1908

O_RDONLY 624, 1190, 1320, 1709, 1906, 1916, 2124, 2935, 3083, 3087
 O_RDWR 624, 1190, 1320, 1563, 1709, 1906, 1916, 1958, 2035, 2124, 2935, 3083, 3087
 O_REALIDS 1191, 1244, 1246, 1908
 O_RSYNC 1191, 1363, 1908
 O_SYNC 129, 1191, 1359, 1363, 1908, 3743
 O_TRUNC 381, 1192, 1328, 1909, 1916, 2935, 3083, 3087
 O_WRONLY 381, 624, 1190, 1320, 1563, 1709, 1906, 1916, 2124, 3083, 3087
 OCBs (Open Control Blocks) 1253, 1255, 1312, 1313, 1315, 1317, 1319, 1373, 2634, 2636, 2644
 allocating 1313
 attaching 1312, 2636
 detaching 1253, 1255, 1315, 2644
 freeing 1255, 1317
 getting 2634
 structure 1319
 unlocking 1373
 off_t, limiting to 32 bits 784, 1691, 3171
 offsetof() 1900
 OHFLOW 3381
 OLD_TIME 987, 3594
 once-initialization 2361
 OOB (out-of-band) data 1301, 1387, 2533, 2536, 2539, 3073
 checking for availability 1301, 1387
 determining if at mark 3073
 sending/receiving 2533, 2536, 2539
 Open Control Blocks, See OCBs
 OPEN_MAX 2903, 3089, 3122
 open(), open64() 381, 1325, 1905, 3558
 resource managers, implementing in 1325
 opendir() 1913
 openfd() 1327, 1330, 1916
 resource managers, implementing in 1327, 1330
 openlog() 1920
 openpty() 1922
 operating system 51, 345, 346, 3571
 name 346, 3571
 release level 345
 target 51
 version 51, 346, 3571
 OPOST 3380
 optarg 904
 opterr 904
 optimization, compiling with 51
 optind 903
 options 903, 963, 977, 1694, 2910
 command-line 903, 977
 parsing 903, 977
 mount, parsing 1694
 socket-level 963, 2910
 optopt 904
 OSIOCGIFADDR 1199
 OSIOCGIFBRDADDR 1199
 OSIOCGIFCONF 1199
 OSIOCGIFDSTADDR 1199
 OSIOCGIFNETMASK 1199
 other scheduling 2223

- out-of-band (OOB) data 1301, 1387, 2533, 2536, 2539, 3073
 - checking for availability 1301, 1387
 - determining if at mark 3073
 - sending/receiving 2533, 2536, 2539
- out16() 1927
- out16s() 1929
- out32() 1931
- out32s() 1933
- out8() 1923
- out8s() 1925
- outbe16() 1927
- outbe32() 1931
- outle16() 1927
- outle32() 1931
- output, formatted 702, 806, 2133, 3616, 3620, 3624, 3647, 3736
- overlapping memory, copying 1618, 3730
- ownership, changing of a file 292, 615, 1502

P

- P_ALL 3663
- P_NOWAIT 3099, 3105, 3111, 3116, 3129, 3134, 3140, 3145
- P_NOWAITO 3099, 3105, 3111, 3116, 3129, 3134, 3140, 3145
- P_OVERLAY 3099, 3105, 3111, 3116, 3129, 3134, 3140, 3145
- P_PGID 3663
- P_PID 3663
- P_WAIT 3099, 3105, 3111, 3116, 3129, 3134, 3140, 3145
- packets 2663, 2664
 - See also ROUTE
 - routing 2664
 - See also ROUTE
- page sizes 908, 909
 - available 909
 - current 908
- pager 2148
- PARENB 3381
- PARMRK 3380
- PARODD 3381
- PARSTK 3381
- partitions, adaptive, See adaptive partitions
- passwd 930, 934, 937, 938, 941
- passwords 385, 524, 526, 654, 911, 930, 932, 934, 937, 938, 941, 971, 975, 2459, 2474, 2894, 2912
 - database 524, 930, 932, 934, 937, 938, 941, 2894
 - closing 524
 - entries, getting for a user 934, 937, 938, 941
 - entries, getting next 930, 932
 - rewinding 2894
- encrypting 385, 2474
- prompting for and reading 911
- shadow database 526, 654, 971, 975, 2459, 2912
 - closing 526
 - entry, reading 654, 971, 975
 - entry, structure 2459
 - entry, writing 2459
 - rewinding 2912
- PATH 300, 569, 592, 2785, 3111, 3116, 3122, 3140, 3145
- PATH_MAX 830, 1853
- pathconf() 1332, 1335, 1936
 - resource managers, implementing in 1332, 1335
- pathfind(), pathfind_r() 1940
- pathmgr_symlink() 1943, 2145
- pathmgr_unlink() 1945
- paths 202, 424, 678, 700, 1000, 1004, 1937, 2531, 2581, 2600, 2639, 3661
 - base name 202
 - directory name 424
 - names 678, 700, 1000, 1004, 1937
 - maximum length 700, 1937
 - patterns, matching 678, 1000, 1004
 - truncating 700, 1937
 - resolving 2531
 - resource managers 2581, 2600, 2639
 - attaching to 2581
 - detaching from 2600
 - getting 2639
 - waiting for 3661
- pathtrust 2145
- pattern matching, See regular expressions
- pause() 1947
- PC Card server 1950, 1952, 1954, 1956, 1958, 1961, 1962
 - arming 1950
 - attaching 1952
 - card insertion/removal, notification of 1950
 - CIS (Card Information Structure), reading 1961
 - detaching 1954
 - locking 1958
 - socket setup information 1956
 - unlocking 1962
- pccard_arm() 1950
- pccard_attach() 1952
- pccard_detach() 1954
- pccard_info() 1956
- pccard_lock() 1958
- pccard_raw_read() 1961
- pccard_unlock() 1962
- PCI 1664, 1964, 1966, 1971, 1975, 1977, 1979, 1982, 1983, 1985, 1988, 1989, 1991, 1993, 1995, 1997, 1999, 2001, 2004, 2006
 - addresses 1971
 - converting 1971
 - testing 1971
 - BIOS, determining if present 1988
 - classes, finding 1979
 - devices 1966, 1977, 1979, 1982, 1989, 1991, 1993, 1995, 1997, 1999, 2001, 2004, 2006
 - attaching 1966
 - configuration, reading 1989, 1991, 1993, 1995
 - configuration, writing 1999, 2001, 2004, 2006
 - detaching 1977
 - finding 1979, 1982
 - rescanning for 1997
 - functions, finding 1979
 - interrupts 1983, 1985
 - mapping 1985
 - routing information 1983
 - memory, sharing 1664
 - server 1964, 1975
 - attaching 1964

PCI (*continued*)
 server (*continued*)
 detaching 1975
 pci_attach_device() 1966
 pci_attach() 1964
 PCI_BAD_REGISTER_NUMBER 1990, 1992, 1994, 1996, 2002, 2004, 2006
 PCI_BUFFER_TOO_SMALL 1990, 1992, 1994, 2002, 2004
 pci_detach_device() 1977
 pci_dev_info 1967
 PCI_DEVICE_NOT_FOUND 1977, 1980, 1982, 1990
 pci_find_class() 1979
 pci_find_device() 1982
 PCI_INIT_ALL 1970
 PCI_INIT_BASE0-PCI_INIT_BASE5 1970
 PCI_INIT_IRQ 1970
 PCI_INIT_ROM 1970
 PCI_IO_ADDR() 1971
 pci_irq_routing_options() 1983
 PCI_IS_IO() 1971
 PCI_IS_MEM() 1971
 pci_map_irq() 1985
 PCI_MASTER_ENABLE 1970
 PCI_MEM_ADDR() 1971
 PCI_PERSIST 1969, 1977
 pci_present() 1988
 pci_read_config() 1989
 pci_read_config16() 1993
 pci_read_config32() 1995
 pci_read_config8() 1991
 pci_rescan_bus() 1997
 PCI_ROM_ADDR() 1971
 PCI_SEARCH_BUSDEV 1970
 PCI_SEARCH_CLASS 1970
 PCI_SEARCH_VEND 1969
 PCI_SEARCH_VENDEV 1970
 PCI_SET_FAILED 1986
 PCI_SHARE 1969
 PCI_SUCCESS 1980, 1982, 1988, 1992, 1994, 1996, 2002, 2004, 2006
 PCI_UNSUPPORTED_FUNCTION 1986
 PCI_USE_MSI 1970
 PCI_USE_MSIX 1970
 pci_write_config() 1999
 pci_write_config16() 2004
 pci_write_config32() 2006
 pci_write_config8() 2001
 pclose() 2007, 2020
 peers, getting names of connected 915
 peers, UNIX-domain 913
 Peripheral Component Interconnect, See PCI
 permissions 289, 612, 2156, 3558
 changing 289, 612
 files, on creation 2156, 3558
 daemons 2156
 perror() 2009
 PF_KEY 1396
 PF_KEY_V2 1396
 PF_ROUTE 1101
 PF_UNSPEC 2492
 physical storage, synchronizing memory with 1825
 PIPE_BUF 3738
 pipe() 2011
 pipes 700, 1937, 2007, 2011, 2019, 2131, 2502, 2523
 bytes, writing atomically 700, 1937
 closing 2007
 creating 2011
 opening 2019
 reading from 2131, 2502, 2523
 pointers, size of void 51
 policy specifications 1406
 poll() 2013
 POLLERR 2014
 pollfd 2013
 POLLHUP 2014
 POLLIN 2014
 POLLNVAL 2014
 POLLOUT 2014
 POLLPRI 2015
 POLLRDBAND 2015
 POLLRDNORM 2015
 POLLWRBAND 2015
 POLLWRNORM 2015
 POOL_FLAG_EXIT_SELF 3387
 POOL_FLAG_USE_SELF 3387
 popen() 2019
 portable code 46
 ports 211, 954, 1059, 1061, 1063, 1065, 1067, 1069, 1672, 1677, 1680, 1686, 1923, 1925, 1927, 1929, 1931, 1933, 2670, 2674
 managing 2674
 privileged 211, 2670
 socket, binding to 211
 socket, getting for 2670
 reading from 1059, 1061, 1063, 1065, 1067, 1069
 serial 1672, 1677, 1680, 1686
 opening 1672
 reading 1677
 script, running on 1680
 writing 1686
 services, finding for 954
 writing to 1923, 1925, 1927, 1929, 1931, 1933
 POSIX 41, 46, 2978, 3310
 See also message queues; semaphores; threads
 signals 2978
 standards 46
 version supported 3310
 See also message queues; semaphores; threads
 POSIX_FADV_DONTNEED 2022
 POSIX_FADV_NOREUSE 2023
 POSIX_FADV_NORMAL 2022
 POSIX_FADV_RANDOM 2022
 POSIX_FADV_SEQUENTIAL 2022
 POSIX_FADV_WILLNEED 2022
 posix_fadvise(), posix_fadvise64() 2022
 posix_fallocate(), posix_fallocate64() 2025
 POSIX_MADV_DONTNEED 2028
 POSIX_MADV_NORMAL 2028
 POSIX_MADV_RANDOM 2028
 POSIX_MADV_SEQUENTIAL 2028
 POSIX_MADV_WILLNEED 2028
 posix_madvise() 2028
 posix_mem_offset() 2030
 posix_mem_offset64() 2030

- posix_memalign() 2033
- posix_openpt() 2035
- POSIX_SPAWN_ALIGN_DEFAULT 2043, 2119
- POSIX_SPAWN_ALIGN_FAULT 2043, 2119
- POSIX_SPAWN_ALIGN_NOFAULT 2043, 2119
- POSIX_SPAWN_ASLR_INVERT 2043, 2119
- POSIX_SPAWN_CHECK_SCRIPT 2043, 2119
- POSIX_SPAWN_DEBUG 2043, 2119
- POSIX_SPAWN_EXEC 2043, 2119
- POSIX_SPAWN_EXPLICIT_CPU 2043, 2076, 2105, 2119
- posix_spawn_file_actions_addclose() 2040, 2048
- posix_spawn_file_actions_adddup2() 2040, 2050
- posix_spawn_file_actions_addopen() 2040, 2052
- posix_spawn_file_actions_destroy() 2041, 2055
- posix_spawn_file_actions_init() 2040, 2057
- posix_spawn_file_actions_t 2040, 2048, 2050, 2052, 2055, 2057
 - close action, adding 2048
 - destroying 2055
 - duplicate action, adding 2050
 - initializing 2057
 - open action, adding 2052
- POSIX_SPAWN_HOLD 2043, 2119
- POSIX_SPAWN_NEWAPP 2043, 2119
- POSIX_SPAWN_NOZOMBIE 2043, 2119
- POSIX_SPAWN_PADDR64_SAFE 2043, 2119
- POSIX_SPAWN_RESETIDS 2042, 2096, 2098, 2119
- POSIX_SPAWN_SEARCH_PATH 2043, 2120
- POSIX_SPAWN_SETCRED 2043, 2066, 2095, 2120
- POSIX_SPAWN_SETND 2043, 2070, 2101, 2120
- POSIX_SPAWN_SETPGROUP 2042, 2074, 2098, 2103, 2119
- POSIX_SPAWN_SETSCHEDPARAM 2042, 2078, 2098, 2107, 2119
- POSIX_SPAWN_SETSCHEDULER 2042, 2078, 2080, 2098, 2107, 2109, 2119
- POSIX_SPAWN_SETSID 2043, 2120
- POSIX_SPAWN_SETSIGDEF 2042, 2082, 2098, 2111, 2119
- POSIX_SPAWN_SETSIGIGN 2043, 2084, 2113, 2120
- POSIX_SPAWN_SETSIGMASK 2042, 2086, 2098, 2115, 2119
- POSIX_SPAWN_SETPART 2043, 2060, 2062, 2072, 2120
- POSIX_SPAWN_SETSTACKMAX 2043, 2088, 2117, 2120
- POSIX_SPAWN_TCSETPGROUP 2043, 2120
- posix_spawn() 2148
- posix_spawn(), posix_spawnnp() 2037, 2142
- posix_spawnattr_addpartid() 2059
- posix_spawnattr_addpartition() 2061
- posix_spawnattr_destroy() 2041, 2064
- posix_spawnattr_getcred() 2066
- posix_spawnattr_getflags() 2042, 2068
- posix_spawnattr_getnode() 2070
- posix_spawnattr_getpartid() 2072
- posix_spawnattr_getpgroup() 2074
- posix_spawnattr_getrunmask() 2076
- posix_spawnattr_getschedparam() 2078
- posix_spawnattr_getschedpolicy() 2080
- posix_spawnattr_getsigdefault() 2082
- posix_spawnattr_getsigignore() 2084
- posix_spawnattr_getsigmask() 2086
- posix_spawnattr_getstackmax() 2088
- posix_spawnattr_getxflags() 2042, 2090
- posix_spawnattr_init() 2041, 2093
- posix_spawnattr_setcred() 2095
- posix_spawnattr_setflags() 2042, 2098
- posix_spawnattr_setnode() 2101
- posix_spawnattr_setpgroup() 2103
- posix_spawnattr_setrunmask() 2105
- posix_spawnattr_setschedparam() 2107
- posix_spawnattr_setschedpolicy() 2109
- posix_spawnattr_setsigdefault() 2111
- posix_spawnattr_setsigignore() 2113
- posix_spawnattr_setsigmask() 2115
- posix_spawnattr_setstackmax() 2117
- posix_spawnattr_setxflags() 2042, 2119
- posix_spawnattr_t 2041
- POSIX_TYPED_MEM_ALLOCATE 2123, 2124
- POSIX_TYPED_MEM_ALLOCATE_CONTIG 2123, 2124
- posix_typed_mem_get_info() 2122
- POSIX_TYPED_MEM_MAP_ALLOCATABLE 2124
- posix_typed_mem_open() 2124
- pow(), powf(), powl() 2129
- powers 2129
- pread(), pread64() 2131
- precision, floating-point 695
- printable, testing a character for 1439, 1447, 1467, 1471
- printf() 2133
- priorities 279, 281, 922, 1892, 2041, 2042, 2078, 2080, 2107, 2109, 2318, 2324, 2329, 2402, 2720, 2723, 2725, 2727, 2738, 2741, 2890, 3094
 - adjusting 1892, 2723
 - getting 922, 2720
 - inheritance 279, 281, 2318, 2324, 2329
 - preventing 281
 - maximum 2725
 - minimum 2727
 - setting 2041, 2042, 2078, 2080, 2107, 2109, 2402, 2738, 2741, 2890, 3094
- prioritized I/O, support for (_PC_ASYNC_IO) 700, 1937
- process groups 552, 558, 564, 575, 581, 587, 597, 917, 919, 1494, 1496, 1790, 2041, 2042, 2043, 2074, 2103, 2176, 2886, 2889, 2907, 2951, 2983, 2987, 3076, 3091, 3092, 3094, 3095, 3349, 3351, 3367, 3658, 3663, 3666
 - changing 2176
 - creating 2886, 2907, 3095
 - devices 2043, 3349, 3367
 - ID, getting 917, 919
 - joining 2886
 - membership, inheriting 552, 558, 564, 575, 581, 587, 2041, 2042, 2074, 2103, 3091
 - pulses, sending 1790
 - remote node 3092
 - session of a controlling terminal 3351
 - setting 2889, 3094
 - signals, sending 597, 1494, 1496, 2951, 2983, 2987, 3076
 - SIGHUP 597
 - SIGURG 3076
 - status of 3658, 3666
 - waiting for 3663
- processes 41, 54, 144, 153, 154, 161, 169, 199, 213, 247, 254, 279, 281, 300, 304, 305, 307, 313, 327, 333, 341, 342, 352, 356, 361, 363, 367, 370, 372, 387, 394, 397, 401, 449, 451, 453, 455, 459, 461, 528, 550, 556, 562, 569, 573,

processes (*continued*)

579, 585, 592, 596, 599, 685, 689, 836, 838,
840, 849, 864, 891, 892, 903, 920, 921, 922,
958, 984, 1148, 1154, 1155, 1159, 1160, 1162,
1164, 1166, 1168, 1171, 1174, 1176, 1178,
1179, 1181, 1183, 1494, 1496, 1549, 1566,
1651, 1653, 1656, 1669, 1828, 1830, 1855,
1867, 1868, 1892, 1947, 2028, 2037, 2039,
2040, 2041, 2042, 2043, 2048, 2050, 2052,
2059, 2061, 2066, 2070, 2072, 2074, 2076,
2078, 2080, 2082, 2084, 2086, 2088, 2095,
2101, 2103, 2105, 2107, 2109, 2111, 2113,
2115, 2117, 2141, 2152, 2154, 2156, 2158,
2159, 2163, 2164, 2168, 2170, 2172, 2178,
2183, 2318, 2324, 2329, 2454, 2482, 2720,
2723, 2725, 2727, 2729, 2736, 2738, 2741,
2743, 2756, 2771, 2851, 2853, 2856, 2858,
2864, 2875, 2889, 2890, 2895, 2897, 2900,
2907, 2917, 2948, 2950, 2970, 2974, 2978,
2979, 2982, 2987, 2995, 2997, 3002, 3007,
3010, 3016, 3020, 3022, 3028, 3089, 3092,
3093, 3094, 3095, 3100, 3106, 3112, 3117,
3122, 3130, 3135, 3141, 3146, 3308, 3309,
3310, 3330, 3409, 3410, 3412, 3413, 3428,
3452, 3471, 3518, 3519, 3550, 3558, 3584,
3586, 3614, 3652, 3656, 3659, 3664, 3667

See also threads

64-bit addressing 2043, 3094

abilities 361, 2141, 2152, 2154

checking 361

controlling 2141

creating 2152

looking up 2154

adaptive partition 2041, 2043, 2059, 2061, 2072,
2756, 2771

getting 2072, 2771

setting 2041, 2043, 2059, 2061, 2756

address space 1651, 1653, 1669, 1828, 1830, 2900

device I/O memory, mapping 1669

limits 2900

locking 1651, 1653

unlocking 1828, 1830

Address Space Layout Randomization (ASLR) 2041, 2043

alarms, scheduling 144, 3550

alignment 2041, 2043, 3094

analyzing 1171

arguments 153, 154, 199, 903, 3308

auxiliary 199

maximum length 3308

number of 153

parsing 903

vector of 154

asynchronous I/O, priority of (`_SC_AIO_PRIO_DELTA_MAX`)
3308

background 394, 2156, 2159, 2164

termination, notification of 2159, 2164

child 2040, 2041, 2043, 2048, 2050, 2052, 2950,
2982, 3089, 3092, 3093, 3094, 3122, 3652,
3656, 3659, 3664, 3667

closing file descriptors in 2040, 2048, 3089, 3093,
3122

processes (*continued*)child (*continued*)

specifying file descriptors for 2040, 2050, 2052,
3089, 3092, 3122

state change, waiting for 3652, 3656, 3659, 3664,
3667

zombie, preventing from becoming 2041, 2043,
2950, 2982, 3094

configurable limits 3308

connections 281, 304, 356, 361, 363, 367, 370, 372

client, information about 304, 356, 361, 363

detaching 281, 367

flags, modifying 370

information about 372

controlling terminal, path name 387

CPU time, maximum 2900

creating 550, 556, 562, 569, 573, 579, 585, 592, 685,
689, 2037, 2043, 3089, 3094, 3100, 3106, 3112,
3117, 3122, 3130, 3135, 3141, 3146

data segment, changing space allocated for 213

debugging 397, 2043, 3094

dynamically linked libraries 449, 451, 453, 455, 459,
461

addresses, translating 449

closing 451

debugging 459

errors 453

opening 455

symbol, getting address of 461

environment 300, 528, 1549, 2875, 2970, 3010

clearing 300

restoring 1549, 2970

saving 2875, 3010

environment variables 838, 2454, 2853, 3584

defining 2454, 2853

deleting 2454, 2853, 3584

getting 838

executable file 341, 342, 2183

base name 2183

file descriptor 341

full path 342

executing 550, 556, 562, 569, 573, 579, 585, 592,
2043, 3094

execution time 313, 333

execution time limit, getting 2736

file-mode creation mask 2156, 3558

daemons 2156

files, maximum per 3309

forking 685, 689

group ID 836, 849, 864, 2039, 2041, 2042, 2043,
2066, 2095, 2851, 2858, 2864, 2895

effective 836, 2039, 2041, 2042, 2043, 2851,
2858, 2895

real 849, 2039, 2041, 2042, 2043, 2066, 2095,
2858, 2895

saved 2858

supplementary 864, 2864

guardian, specifying 2172

holding for debugging 2041, 2043, 3094

I/O privileges, requesting 247, 254, 1148, 1155, 1160,
1162, 1164, 1166, 1174, 1176, 1179, 1181, 2979,
3409, 3410, 3518, 3519

processes (*continued*)

- ID, getting 920, 921
- interrupts 1148, 1154, 1159, 1162, 1164, 1166, 1168, 1174, 1176, 1178, 1179, 1181, 1183
 - disabling 1164
 - enabling 1166
 - events 1154, 1162
 - handlers 1148, 1162, 1174, 1179
 - handlers, idle 1168
 - masking 1176, 1181
 - status of 1159, 1178
 - waiting for 1183
- maximum per real user ID 2900, 3308
- memory 1656, 2028
 - advising about use of 2028
 - sharing 1656
- message channels, attaching to 352
- name 342, 2183
- node, running on 2041, 2043, 2070, 2101
- parent 921, 3614
 - blocking 3614
 - ID, getting 921
- priority 279, 281, 922, 1892, 2041, 2042, 2078, 2080, 2107, 2109, 2318, 2324, 2329, 2720, 2723, 2725, 2727, 2738, 2741, 2890, 3094
 - adjusting 1892, 2723
 - getting 922, 2720
 - inheritance 279, 2318, 2324, 2329
 - inheritance, preventing 281
 - maximum 2725
 - minimum 2727
 - setting 2041, 2042, 2078, 2080, 2107, 2109, 2738, 2741, 2890, 3094
- processor affinity 2041, 2043, 2076, 2105, 3094, 3095, 3412, 3413
- program entry function 1566
- resident set size, maximum 2900
- scheduling parameters 2041, 2042, 2078, 2080, 2107, 2109, 3094
- scheduling policy 2041, 2042, 2078, 2080, 2107, 2109, 2729, 2741, 3094
 - getting 2729
 - setting 2041, 2042, 2078, 2080, 2107, 2109, 2741, 3094
- sessions 958, 2889, 2907
- set-group ID 2041, 3309
- set-user ID 2041, 3309
- SIGALRM sending to 3428, 3452, 3550
- signal actions, inheriting 2041, 2042, 2074, 2082, 2084, 2086, 2103, 2111, 2113, 2115
- signals 1494, 1496, 1947, 2482, 2948, 2974, 2978, 2987, 2995, 2997, 3002, 3007, 3016, 3020, 3022, 3310
 - actions for 2948, 2974, 2978
 - information about 3022
 - maximum outstanding (`_SC_SIGQUEUE_MAX`) 3310
 - pending 3002
 - queuing 3007
 - raising 2482
 - sending 1494, 1496, 2987
 - suspending until delivered 2995
 - waiting for 1947, 2997, 3016, 3020, 3022

processes (*continued*)

- spawning 2037, 3089, 3100, 3106, 3112, 3117, 3122, 3130, 3135, 3141, 3146
- spawning and blocking 3614
- stack size, setting maximum for 2041, 2043, 2088, 2117, 3095
- supplementary group IDs, maximum 3309
- suspending 401, 1855, 1867, 1868, 3028, 3586, 3656
- system commands, executing 3330
- system-wide events 2158, 2163, 2168, 2170, 2178
 - notification of 2158, 2163, 2168, 2178
 - triggering 2170
- terminating 54, 161, 169, 596, 599
 - diagnostics 161
 - functions to be called, registering 169
- time 305, 307, 327
 - clock ID 307, 327
 - clock ticks 305
- time-accounting information 3471
- user ID 840, 984, 2039, 2041, 2042, 2043, 2066, 2095, 2856, 2897, 2917
 - effective 840, 2039, 2041, 2042, 2043, 2856, 2897, 2917
 - real 984, 2039, 2041, 2042, 2043, 2066, 2095, 2897, 2917
 - saved 2917
- user name, getting 891, 892
- yielding 2743
- zombies, preventing children from becoming 2041, 2043, 2950, 2982, 3094
 - See also threads

ProcessInfo_t 2945

- processor affinity 2041, 2043, 2076, 2105, 3094, 3095, 3412, 3413
- procmgr_ability_create() 2142, 2152
- procmgr_ability_lookup() 2154
- procmgr_ability() 2141
- PROCMGR_ADN_NONROOT 2150
- PROCMGR_ADN_ROOT 361, 2150, 2152
- PROCMGR_AID_ABLE_CREATE 2142, 2153
- PROCMGR_AID_APS_ROOT 2142
- PROCMGR_AID_CHILD_NEWAPP 2142
- PROCMGR_AID_CLOCKPERIOD 330, 2142
- PROCMGR_AID_CLOCKSET 320, 323, 334, 2143, 2915
- PROCMGR_AID_CONFSET 344, 2143
- PROCMGR_AID_CONNECTION 370, 1790, 2143, 3292
- PROCMGR_AID_CPUMODE 2143
- PROCMGR_AID_DEFAULT_TIMER_TOLERANCE 2143, 2178
- PROCMGR_AID_EOL 2150
- PROCMGR_AID_EVENT 2143, 2170
- PROCMGR_AID_FORK 685, 2143
- PROCMGR_AID_GETID 917, 958, 2143
- PROCMGR_AID_INTERRUPT 1148, 1155, 1168, 1171, 2144
- PROCMGR_AID_IO 247, 254, 1060, 1062, 1064, 1066, 1068, 1070, 1148, 1155, 1162, 1164, 1166, 1171, 1174, 1176, 1179, 1181, 1667, 1924, 1926, 1928, 1930, 1932, 1934, 2144, 2979, 3410, 3518, 3519
- PROCMGR_AID_KEYDATA 1748, 2144
- PROCMGR_AID_MAP_FIXED 1660, 1670, 2144
- PROCMGR_AID_MEM_ADD 2144
- PROCMGR_AID_MEM_GLOBAL 2144, 2925, 2931
- PROCMGR_AID_MEM_LOCK 1651, 1654, 2144

PROCMGR_AID_MEM_PEER 2144
 PROCMGR_AID_MEM_PHYS 1662, 1666, 1669, 2144, 2926, 2932
 PROCMGR_AID_MEM_SPECIAL 2144, 2933
 PROCMGR_AID_PATH_TRUST 2145
 PROCMGR_AID_PATHSPACE 1943, 2145, 2581
 PROCMGR_AID_PGRP 2145, 2176, 2889, 2907
 PROCMGR_AID_PRIORITY 2145, 2322, 2349, 2351, 2401, 2403, 2739, 2741, 2781, 2890, 3287, 3306
 PROCMGR_AID_PROT_EXEC 455, 633, 1656, 1668, 1699, 2146
 PROCMGR_AID_PUBLIC_CHANNEL 281, 2146
 PROCMGR_AID_QNET 2146
 PROCMGR_AID_RCONSTRAINT 2146
 PROCMGR_AID_REBOOT 2146, 3319
 PROCMGR_AID_RLIMIT 2146, 2900
 PROCMGR_AID_RSRCDBMGR 2146, 2675, 2686
 PROCMGR_AID_RUNSTATE 2146, 3321
 PROCMGR_AID_RUNSTATE_BURST 2146, 3322
 PROCMGR_AID_SCHEDULE 2147, 2300, 2401, 2775, 2781
 PROCMGR_AID_SESSION 2147, 2176, 3367
 PROCMGR_AID_SETGID 2147, 2851, 2864
 PROCMGR_AID_SETUID 2147, 2917
 PROCMGR_AID_SIGNAL 1495, 2147, 2310, 2408, 2949, 2974, 2978, 2987, 2992, 3005, 3008
 PROCMGR_AID_SPAWN 550, 557, 563, 569, 573, 580, 585, 592, 2148, 3090, 3100, 3106, 3112, 3117, 3123, 3130, 3135, 3141, 3146
 PROCMGR_AID_SPAWN_SETGID 2148
 PROCMGR_AID_SPAWN_SETUID 2148
 PROCMGR_AID_SWAP 2148
 PROCMGR_AID_TIMER 2148, 3428, 3433, 3435, 3438, 3452, 3457
 PROCMGR_AID_TRACE 2149, 3513, 3518, 3519
 PROCMGR_AID_UMASK 2149, 3558
 PROCMGR_AID_V86 1186, 2149
 PROCMGR_AID_WAIT 2149, 3652, 3656, 3659, 3664, 3667
 PROCMGR_AOP_ALLOW 2149
 PROCMGR_AOP_DENY 2149
 PROCMGR_AOP_INHERIT_NO 2150
 PROCMGR_AOP_INHERIT_YES 2150
 PROCMGR_AOP_LOCK 2150
 PROCMGR_AOP_SUBRANGE 2142, 2149
 PROCMGR_DAEMON_KEEPUMASK 2156
 PROCMGR_DAEMON_NOCHDIR 2156
 PROCMGR_DAEMON_NOCLOSE 2156
 PROCMGR_DAEMON_NODEVNULL 2156
 procmgr_daemon() 2156
 PROCMGR_EVENT_CONFSTR 2159, 2164
 PROCMGR_EVENT_DAEMON_DEATH 2159, 2164
 procmgr_event_notify_add() 2163
 procmgr_event_notify_delete() 2168
 procmgr_event_notify() 2158
 PROCMGR_EVENT_PATHSPACE 2159, 2164
 PROCMGR_EVENT_SYNC 2159, 2164
 PROCMGR_EVENT_SYSCONF 2159, 2164
 PROCMGR_EVENT_TOD 2160, 2164
 procmgr_event_trigger() 2143, 2170
 procmgr_guardian() 2172
 PROCMGR_SESSION_SETPGRP 2176
 PROCMGR_SESSION_SETSID 2176
 PROCMGR_SESSION_SIGNAL_LEADER 2176
 PROCMGR_SESSION_SIGNAL_PGRP 2176
 PROCMGR_SESSION_SIGNAL_PID 2176
 PROCMGR_SESSION_TCSETSID 2176
 procmgr_session() 2145, 2147, 2176
 procmgr_timer_tolerance() 2143, 2178
 procmgr_value_current() 2180
 PROCMGR_VALUE_FREE_MEM 2181
 procmgr_value_notify_add() 2182
 PROCMGR_VALUE_PRIVATE_MEM 2181
 PROCMGR_VALUE_TRIGGER_DOWN 2181
 PROCMGR_VALUE_TRIGGER_UP 2181
 procnto 2145
 program entry function 1566
 PROMPT_CANCEL 2946
 PROMPT_KILL 2946
 PROMPT_SKIP 2946
 PROMPT_WAIT 2946
 PROT_EXEC 633, 1295, 1656, 1668, 1699, 2146
 PROT_NOCACHE 1295, 1657, 1668, 1699, 2927
 PROT_NONE 1295, 1657, 1669, 1699
 PROT_READ 1295, 1657, 1669, 1699
 PROT_WRITE 1295, 1657, 1669, 1699
 protocols 523, 924, 926, 928, 1045, 1047, 1100, 1391, 1396, 1412, 2184, 2892, 3359, 3553, 3577
 database 523, 924, 926, 928, 2184, 2892
 closing 523
 entries, getting 924, 926, 928
 entry structure 2184
 opening 2892
 ICMP (Internet Control Message Protocol) 1045
 ICMP6 (Internet Control Message Protocol v6) 1047
 INET6 (Internet protocol v6 family) 1100
 interprocess communication 3577
 IP (Internet Protocol) 1391
 IP6 (Internet Protocol v6) 1412
 IPsec (secure Internet Protocol) 1396
 TCP (Transmission Control Protocol) 3359
 UDP (User Datagram Protocol) 3553
 protoent 2184
 proxy server (SOCKS) 3764
 pseudo-random numbers 468, 531, 1133, 1492, 1505, 1551, 1730, 1894, 2484, 2486, 2487, 2787, 2913, 3155, 3157, 3158
 double 468, 531
 int 2484, 2486
 long 1492, 1551, 1730, 1894, 2487
 nonnegative 1551, 1894
 signed 1492, 1730, 2487
 seed, setting 2787, 3155, 3157, 3158
 sequence, initializing 3157
 state 1133, 1505, 2913
 initializing 1133, 1505
 switching 2913
 pseudo-ttys 689, 1009, 1547, 1922, 2035, 2436, 2438, 3582
 granting access to the slave device 1009
 name, getting 2436, 2438
 opening 689, 1922, 2035
 preparing for a login 689, 1547
 unlocking 3582
 PSIZ 151
 pthread_abort() 2185

PTHREAD_ABORTED 2185
pthread_atfork() 2187
pthread_attr_destroy() 2189
pthread_attr_getdetachstate() 2191
pthread_attr_getguardsize() 2193
pthread_attr_getinheritsched() 2195
pthread_attr_getschedparam() 2197
pthread_attr_getschedpolicy() 2199
pthread_attr_getscope() 2201
pthread_attr_getstack() 2203
pthread_attr_getstackaddr() 2205
pthread_attr_getstacklazy() 2207
pthread_attr_getstackprealloc() 2209
pthread_attr_getstacksize() 2211
pthread_attr_init() 2213
pthread_attr_setdetachstate() 2215
pthread_attr_setguardsize() 2217
pthread_attr_setinheritsched() 2219
pthread_attr_setschedparam() 2221
pthread_attr_setschedpolicy() 2223
pthread_attr_setscope() 2225
pthread_attr_setstack() 2227
pthread_attr_setstackaddr() 2229
pthread_attr_setstacklazy() 2231
pthread_attr_setstackprealloc() 2233
pthread_attr_setstacksize() 2235
pthread_attr_t 2213, 2284
pthread_barrier_destroy() 2237
pthread_barrier_init() 2239
PTHREAD_BARRIER_INITIALIZER() 2240
PTHREAD_BARRIER_SERIAL_THREAD 2241
pthread_barrier_wait() 2241
pthread_barrierattr_destroy() 2243
pthread_barrierattr_getpshared() 2245
pthread_barrierattr_init() 2247
pthread_barrierattr_setpshared() 2249
PTHREAD_CANCEL 3400
PTHREAD_CANCEL_ASYNCHRONOUS 2286, 2394, 3404
PTHREAD_CANCEL_DEFERRED 2213, 2286, 2394, 3404, 3405
PTHREAD_CANCEL_DISABLE 2285, 2392
PTHREAD_CANCEL_ENABLE 2213, 2285, 2392, 3405
pthread_cancel() 2251
PTHREAD_CANCELED 2303, 2434
pthread_cleanup_pop() 2253
pthread_cleanup_push() 2255
pthread_cond_broadcast() 2257
pthread_cond_destroy() 2259
pthread_cond_init() 2261
PTHREAD_COND_INITIALIZER 2261, 3280, 3282
pthread_cond_signal() 2263
pthread_cond_timedwait() 2265
pthread_cond_wait() 2269
pthread_condattr_destroy() 2272
pthread_condattr_getclock() 2274
pthread_condattr_getpshared() 2276
pthread_condattr_init() 2278
pthread_condattr_setclock() 2281
pthread_condattr_setpshared() 2282
PTHREAD_CREATE_DETACHED 2215, 3403
PTHREAD_CREATE_JOINABLE 2213, 2215, 3403
pthread_create() 2285
PTHREAD_DESTRUCTOR_ITERATIONS 2306
pthread_detach() 2288
pthread_equal() 2290
pthread_exit() 2292
PTHREAD_EXPLICIT_SCHED 2219, 2221, 2223, 3403, 3405
pthread_getconcurrency() 2294
pthread_getcpuclockid() 2295
pthread_getname_np() 2297
pthread_getschedparam() 2147, 2299
pthread_getspecific() 2301
PTHREAD_INHERIT_SCHED 2213, 2219
pthread_join() 2303
pthread_key_create() 2305
pthread_key_delete() 2308
pthread_kill() 2148, 2310
PTHREAD_MULTISIG_ALLOW 2286, 3404
PTHREAD_MULTISIG_DISALLOW 2286, 3404
PTHREAD_MUTEX_DEFAULT 2357
pthread_mutex_destroy() 2312
PTHREAD_MUTEX_ERRORCHECK 2357
pthread_mutex_getprioceiling() 2314
pthread_mutex_init() 2316
PTHREAD_MUTEX_INITIALIZER 2317
pthread_mutex_lock() 2318
PTHREAD_MUTEX_NORMAL 2357
PTHREAD_MUTEX_RECURSIVE 2357
pthread_mutex_setprioceiling() 2145, 2322
pthread_mutex_timedlock(),
pthread_mutex_timedlock_monotonic() 2324
pthread_mutex_trylock() 2327
pthread_mutex_unlock() 2329
pthread_mutex_wakeup_np() 2331
pthread_mutexattr_destroy() 2333
pthread_mutexattr_getprioceiling() 2335
pthread_mutexattr_getprotocol() 2337
pthread_mutexattr_getpshared() 2339
pthread_mutexattr_getrecursive() 2341
pthread_mutexattr_gettype() 2343
pthread_mutexattr_getwakeup_np() 2345
pthread_mutexattr_init() 2347
pthread_mutexattr_setprioceiling() 2145, 2349
pthread_mutexattr_setprotocol() 2145, 2352
pthread_mutexattr_setpshared() 2353
pthread_mutexattr_setrecursive() 2355
pthread_mutexattr_settype() 2358
pthread_mutexattr_setwakeup_np() 2359
PTHREAD_ONCE_INIT 2361
pthread_once() 2361
PTHREAD_PRIO_INHERIT 2337, 2347, 2351, 3305
PTHREAD_PRIO_NONE 2338, 2351
PTHREAD_PRIO_PROTECT 2338, 2351, 3305
PTHREAD_PROCESS_PRIVATE 2249, 2278, 2353, 2389, 2424
PTHREAD_PROCESS_SHARED 2249, 2278, 2282, 2353, 2389, 2424
PTHREAD_RECURSIVE_DISABLE 2341, 2347, 2355
PTHREAD_RECURSIVE_ENABLE 2341, 2355
PTHREAD_RMUTEX_INITIALIZER 2317
pthread_rwlock_destroy() 2364
pthread_rwlock_init() 2366
PTHREAD_RWLOCK_INITIALIZER 2367
pthread_rwlock_rdlock() 2369

pthread_rwlock_timedrdlock() 2371
 pthread_rwlock_timedwrlock() 2373
 pthread_rwlock_tryrdlock() 2375
 pthread_rwlock_trywrlock() 2377
 pthread_rwlock_unlock() 2379
 pthread_rwlock_wrlock() 2381
 pthread_rwlockattr_destroy() 2383
 pthread_rwlockattr_getpshared() 2385
 pthread_rwlockattr_init() 2387
 pthread_rwlockattr_setpshared() 2389
 PTHREAD_SCOPE_PROCESS 3403
 PTHREAD_SCOPE_SYSTEM 2213, 2225, 3403
 pthread_self() 2391
 pthread_setcancelstate() 2392
 pthread_setcanceltype() 2394
 pthread_setconcurrency() 2396
 pthread_setname_np() 2398
 pthread_setschedparam() 2145, 2147, 2400
 pthread_setschedprio() 2145, 2402
 pthread_setspecific() 2405
 pthread_sigmask() 2148, 2407
 pthread_sleepon_broadcast() 2409
 pthread_sleepon_lock() 2411
 pthread_sleepon_signal() 2413
 pthread_sleepon_timedwait() 2415
 pthread_sleepon_unlock() 2418
 pthread_sleepon_wait() 2419
 pthread_spin_destroy() 2422
 pthread_spin_init() 2424
 pthread_spin_lock() 2426
 pthread_spin_trylock() 2428
 pthread_spin_unlock() 2430
 PTHREAD_STACK_LAZY 2231
 PTHREAD_STACK_MIN 2227, 2229, 2230, 2235, 2236, 3404, 3407
 PTHREAD_STACK_NOTLAZY 2231
 pthread_testcancel() 2432
 pthread_timedjoin(), pthread_timedjoin_monotonic() 2433
 PTHREAD_WAKEUP_DISABLE 2359
 PTHREAD_WAKEUP_ENABLE 2331, 2359
 ptsname_r() 2438
 ptsname() 2436
 pulse_attach() 2443
 pulse_detach() 2445
 pulses 1766, 1769, 1790, 2440, 2443, 2445, 2756, 2771, 2962
 adaptive partition 2756, 2771
 getting 2771
 setting 2756
 compression 1790
 dispatch interface 2443, 2445
 attaching 2443
 detaching 2445
 priority of a receiving thread 2962
 queueing 1790
 receiving 1766, 1769
 sending 1790
 structure 2440
 punctuation, testing a character for 1449, 1473
 putc_unlocked() 2449
 putc() 2447
 putchar_unlocked() 2453

putchar() 2451
 putenv() 570, 592, 2454, 3101, 3113, 3130, 3142
 puts() 2457
 putspent() 2459
 pututline() 2462
 putw() 2464
 putwc() 2466, 2468
 pwrite(), pwrite64() 2470

Q

Qnet 142
 asynchronous I/O and 142
 QNX 4 classification 48
 QNX Neutrino classification 48
 qnx_crypt() 2474
 QoS (Quality of Service) 1878
 qsort() 2476
 qtime 3327
 Quality of Service (QoS) 1878
 QUERY 2567
 queues 1145, 2559
 inserting elements 1145
 removing elements 2559
 quick sort 2476
 quotients 447, 1508
 integer 447
 long and long long integers 1508

R

R_OK 61, 488
 Raccept() 2480
 radix-independent exponents 1545, 2705, 2707
 raise() 1549, 2482
 RAND_MAX 2484, 2486
 rand_r() 2486
 rand() 2484
 random numbers 468, 531, 1133, 1492, 1505, 1551, 1730, 1894, 2484, 2486, 2487, 2787, 2913, 3155, 3157, 3158
 double 468, 531
 int 2484, 2486
 long 1492, 1551, 1730, 1894, 2487
 nonnegative 1551, 1894
 signed 1492, 1730, 2487
 seed, setting 2787, 3155, 3157, 3158
 sequence, initializing 3157
 state 1133, 1505, 2913
 initializing 1133, 1505
 switching 2913
 random() 2487
 raw input mode 700, 1937, 2510, 2511
 buffer 700, 1937
 conditions for input request 2510
 FORWARD qualifier 2511
 MIN qualifier 2510
 TIME qualifier 2510
 TIMEOUT qualifier 2510
 Rbind() 2489
 rcmd() 3766
 rcmd(), rcmd_af() 2491

- Rconnect() 2494
- RCT (resource constraint threshold) 3309
- rcvid (receive identifier) 1761, 1772, 1817
 - checking validity of 1817
- rdchk() 2496
- re_comp() 2498
- re_exec() 2500
- read-write locks 2364, 2366, 2369, 2371, 2373, 2375, 2377, 2379, 2381, 2383, 2385, 2387, 2389, 3289, 3305
 - attributes 2383, 2385, 2387, 2389
 - creating 2387
 - destroying 2383
 - process-shared 2385, 2389
 - destroying 2364, 3289
 - initializing 2366, 3305
 - locking 2369, 2371, 2373, 2375, 2377, 2381
 - for reading 2369, 2371, 2375
 - for writing 2373, 2377, 2381
 - unlocking 2379
- read() 1225, 1337, 1340, 2502, 2510
 - resource managers, implementing in 1337, 1340
- readblock() 2506
- readcond() 2509
- readdir_r() 2518
- readdir() 1913
- readdir(), readdir64() 2514
- readlink() 1343, 2520
 - resource managers, implementing in 1343
- readv() 2523
- realloc() 2527
- realpath() 2531
- realtime timers 316, 1857, 1859, 1861, 1863, 1865, 3428, 3431, 3433, 3435, 3437, 3439, 3461
 - busy-waiting 1857, 1861, 1863, 1865
 - calibrating 1859
 - creating 3428
 - destroying 3431
 - expiry status 3433
 - overruns 3435
 - setting 3461
 - threads 316
 - time 3437, 3439
 - getting 3437
 - setting 3439
- rebooting 3319
- rebooting on bankruptcy 2749
- receive identifier 1761, 1772
- receiving 1760, 1766, 1769, 1771, 2534, 2537, 2540
 - messages 1760, 1771, 2534, 2537, 2540
 - from a socket 2534, 2537, 2540
 - pulses 1766, 1769
- recv() 2534
- recvfrom() 2537
- recvmsg() 2540
- REG_EXTENDED 2542
- REG_ICASE 2542
- REG_NEWLINE 2542
- REG_NOSUB 2542, 2550
- REG_NOTBOL 2549
- REG_NOTEOL 2549
- regcomp() 2543
- regerror() 2547
- regex_t 2542
- regexec() 2550
- regfree() 2552
- registers 325, 693, 1185, 1666, 1834, 1989, 1991, 1993, 1995, 1999, 2001, 2004, 2006
 - devices 1666, 1834, 1989, 1991, 1993, 1995, 1999, 2001, 2004, 2006
 - access to, gaining and relinquishing 1666, 1834
 - reading 1989, 1991, 1993, 1995
 - writing 1999, 2001, 2004, 2006
 - floating-point exceptions 693
 - real-mode software interrupts 1185
 - TSC (Time Stamp Counter) 325
- regmatch_t 2550
- regular expressions 2498, 2500, 2543, 2545, 2547, 2550, 2552
 - basic 2543
 - compiling 2498, 2543
 - errors, explaining 2547
 - extended 2545
 - freeing 2552
 - string, comparing to 2500, 2550
- remainder(), remainderf(), remainderl() 2554
- remainders 447, 1508, 2554
 - floating point 2554
 - integer 447
 - long and long long integers 1508
- remote hosts 1419, 2491, 2668, 2696
 - commands, executing on 2491, 2668
 - identity, checking 1419, 2696
- remove() 2556
- remque() 2559
- rename() 2560
- RES_DEBUG 2563
- RES_DEFNAMES 2564, 2576
- RES_DNSRCH 2564, 2576
- RES_INIT 2564
- res_init() 2563
- res_mkquery() 2567
- res_query() 2570
- res_querydomain() 2573
- RES_RECURSE 2564
- res_search() 2576
- res_send() 2578
- RES_STAYOPEN 2564
- RES_USEVC 2564
- residue, floating point 675
- resmgr_attach() 2145, 2581
- resmgr_attr_t 2583
- resmgr_block() 2589
- resmgr_connect_funcs_t 2592
- resmgr_context_alloc() 2594
- resmgr_context_free() 2596
- resmgr_context_t 2598
- resmgr_detach() 2600
- resmgr_devino() 2603
- RESMGR_FLAG_ATTACH_LOCAL 2583
- RESMGR_FLAG_ATTACH_OTHERFUNC 2583
- RESMGR_FLAG_CROSS_ENDIAN 2583
- RESMGR_FLAG_NO_DEFAULT_FUNC (not implemented) 2583
- RESMGR_FLAG_RCM 2583

resmgr_handle_grow() 2605
 resmgr_handle_tune() 2606
 resmgr_handler() 2608
 resmgr_io_funcs_t 2611
 resmgr_iofuncs() 2614
 resmgr_msg_again() 2616
 resmgr_msgread() 2618
 resmgr_msgreadv() 2620
 resmgr_msgreply() 2622, 2623, 2625
 resmgr_msgreplyv() 2625, 2626
 resmgr_msgwrite() 2628
 resmgr_msgwritev() 2630
 resmgr_ocb() 2634
 resmgr_open_bind() 2636
 resmgr_pathname() 2639
 resmgr_unbind() 2644
 resolv.conf, contents of 345
 resolver routines 465, 466, 1012, 1017, 1024, 2563, 2567,
 2570, 2573, 2576, 2578
 errors 1012, 1017, 1024
 initializing 2563
 Internet domain names 465, 466
 compressing 465
 expanding 466
 options 2563
 queries 2567, 2570, 2573, 2576, 2578
 resource constraint mode 2583
 resource constraint threshold (RCT) 3309
 resource database manager 2674, 2681, 2684, 2686, 2688,
 2691, 2693
 about 2674
 device numbers 2688, 2691
 attaching 2688
 detaching 2691
 resources 2674, 2681, 2684, 2686, 2693
 creating 2681
 destroying 2684
 querying 2693
 reserving 2674
 returning 2686
 resource managers 352, 406, 1189, 1195, 1196, 1206,
 1211, 1212, 1215, 1218, 1219, 1222, 1223,
 1226, 1228, 1231, 1234, 1237, 1239, 1242,
 1245, 1247, 1248, 1249, 1251, 1253, 1255,
 1257, 1261, 1264, 1267, 1268, 1271, 1274,
 1276, 1279, 1281, 1283, 1284, 1286, 1289,
 1292, 1294, 1298, 1302, 1306, 1308, 1312,
 1313, 1315, 1317, 1319, 1322, 1325, 1327,
 1330, 1332, 1335, 1337, 1340, 1343, 1347,
 1351, 1353, 1355, 1358, 1360, 1362, 1364,
 1366, 1368, 1371, 1373, 1375, 1378, 1380,
 1383, 1386, 2144, 2581, 2589, 2592, 2594,
 2596, 2598, 2600, 2603, 2605, 2606, 2608,
 2611, 2614, 2616, 2618, 2620, 2622, 2625,
 2628, 2630, 2632, 2634, 2636, 2639, 2641,
 2643, 2644, 3309
 access, checking 1231
 arming for notification 1386
 attaching 2581
 attributes 1218, 1219, 1222, 1223, 1226, 1228,
 1284, 1364
 initializing 1218

resource managers (*continued*)

attributes (*continued*)

 locking 1219, 1223, 1226, 1284
 structure 1222
 time members, updating 1364
 unlocking 1228
 clients 1245, 1247, 1248, 1366, 1368
 information about 1245, 1247, 1248
 unblocking 1366, 1368
 connect functions 1271, 1325, 2592
 default values, setting 1271
 open 1325
 connect messages 1189, 1195, 1196
 file type reply 1195
 link reply 1196
 structure 1189
 connection IDs 352, 3309
 resource constrain threshold 3309
 context 2594, 2596, 2598
 allocating 2594
 freeing 2596
 structure 2598
 database, expanding capacity 2605
 detaching 2600
 device number, getting 2603
 device-control commands 406
 file descriptors, mapping to OCBs 2606
 function tables, initializing 1271
 helper functions 1234, 1239, 1249, 1257, 1267,
 1274, 1286, 1292, 1294, 1322, 1327, 1332,
 1340, 1343, 1347, 1351, 1353, 1362, 1371,
 1375, 1383
 chmod 1234
 chown 1239
 close 1249
 devctl 1257
 fdinfo 1267
 link 1274
 lseek 1286
 mknod 1292
 mmap 1294
 open 1322
 openfd 1327
 pathconf 1332
 read 1340
 readlink 1343
 rename 1347
 space 1351
 stat 1353
 sync 1362
 unlink 1371
 utime 1375
 write 1383
 I/O functions 1206, 1212, 1215, 1237, 1242, 1251,
 1255, 1261, 1264, 1268, 1271, 1281, 1284,
 1289, 1298, 1330, 1335, 1337, 1355, 1360,
 1368, 1373, 1378, 1380, 2611, 2614
 ACLs 1212, 1215
 chmod 1237
 chown 1242
 client connection, getting for 2614
 close 1251

- resource managers (*continued*)
 - I/O functions (*continued*)
 - default values, setting 1271
 - devctl 1206, 1261, 1264
 - fdinfo 1268
 - lock 1281
 - lseek 1289
 - mmap 1298
 - OCB, close 1255
 - OCB, lock 1284
 - OCB, unlock 1373
 - openfd 1330
 - pathconf 1335
 - read 1337
 - stat 1355
 - structure 2611
 - sync 1360
 - unblock 1368
 - utime 1378
 - write 1380
 - inode number, getting 2603
 - iov_t 2632, 2641
 - filling 2641
 - getting 2632
 - locks 1276, 1279, 1283
 - messages 2589, 2608, 2616, 2618, 2620, 2622, 2625, 2628, 2630
 - blocking while waiting for 2589
 - handling 2608
 - processing again 2616
 - reading 2618, 2620
 - replying 2622, 2625
 - writing 2628, 2630
 - notification 1302, 1306, 1308, 1386
 - arming for 1386
 - installing, polling, and removing 1302
 - removing for a client 1306
 - triggering 1308
 - Open Control Block (OCB) 1253, 1312, 1313, 1315, 1317, 1319, 2606, 2634, 2636, 2644
 - allocating 1313
 - attaching 1312, 2636
 - detaching 1253, 1315, 2644
 - file descriptors, mapping to 2606
 - freeing 1317
 - getting 2634
 - structure 1319
 - path 2581, 2600, 2639
 - attaching to 2581
 - detaching from 2600
 - getting 2639
 - PROCMBR_AID_KEYDATA, retaining 2144
 - server attributes, getting 1211
 - status, returning 2643
 - synchronization, checking to see if required 1358
 - threads in 1223
- resource-constrained mode 3412
- resources, system 835, 943, 945, 2674, 2681, 2684, 2686, 2693, 2899
 - creating 2681
 - destroying 2684
- resources, system (*continued*)
 - limits 835, 943, 2899
 - getting 835, 943
 - setting 2899
 - querying 2693
 - reserving 2674
 - returning 2686
 - usage, getting 945
- rewind() 2646
- rewinddir() 2648
- Rgetsockname() 2650
- rindex() 2652
- rint(), rintf(), rintl() 2654
- RLIM_INFINITY 2900, 2903
- RLIM_SAVED_CUR 2903
- RLIM_SAVED_MAX 2903
- RLIMIT_AS 2900
- RLIMIT_CORE 2900
- RLIMIT_CPU 2900
- RLIMIT_DATA 2900
- RLIMIT_FREEMEM 2900
- RLIMIT_MEMLOCK 2900
- RLIMIT_NOFILE 835, 2900, 2903
- RLIMIT_NPROC 2900
- RLIMIT_NTHR 2900
- RLIMIT_OFILE 2900
- RLIMIT_RSS 2900
- RLIMIT_STACK 2900
- RLIMIT_VMEM 2900
- Rlisten() 2656
- rmdir() 2658
- RMSK_CLR() 3413
- RMSK_ISSET() 3414
- RMSK_SET() 3413
- RMSK_SIZE() 3413
- root directory, changing 295
- roots 258, 3153
 - cube 258
 - square 3153
- round-robin scheduling 2223
- round(), roundf(), roundl() 2661
- rounding 260, 671, 697, 2654, 2661
 - floating point 2654
 - integers 260, 671, 2661
 - mode, floating-point 697
- ROUTE (system packet forwarding database) 2663
- Rrcmd() 2668
- rresvport(), rresvport_af() 2670
- Rselect() 2673
- rshd 2491
- rsrc_alloc_t 2681
- rsrc_request_t 2675
- rsrddbmgr_attach() 2674
- rsrddbmgr_create() 2681
- rsrddbmgr_destroy() 2684
- rsrddbmgr_detach() 2686
- rsrddbmgr_devno_attach() 2688
- rsrddbmgr_devno_detach() 2691
- RSRCDBMGR_DMA_CHANNEL 2676, 2682, 2694
- RSRCDBMGR_FLAG_ALIGN 2676
- RSRCDBMGR_FLAG_LIST 2676
- RSRCDBMGR_FLAG_NAME 2676, 2682

RSRCEBMBGR_FLAG_NOREMOVE 2682
 RSRCEBMBGR_FLAG_RANGE 2676
 RSRCEBMBGR_FLAG_RSVP 2682
 RSRCEBMBGR_FLAG_SHARE 2676
 RSRCEBMBGR_FLAG_TOPDOWN 2676
 RSRCEBMBGR_IO_PORT 2676, 2682, 2694
 RSRCEBMBGR_IRQ 2676, 2682, 2694
 RSRCEBMBGR_MEMORY 2676, 2682, 2694
 RSRCEBMBGR_PCI_MEMORY 2676, 2682, 2694
 rsrcdbmgr_query_name() 2693
 RSRCEBMBGR_IRQ 2682
 rt_metrics 2665
 rt_msghdr 2665
 RTA_AUTHOR 2666
 RTA_BRD 2666
 RTA_DST 2666
 RTA_GATEWAY 2666
 RTA_GENMASK 2666
 RTA_IFA 2666
 RTA_IFP 2666
 RTA_NETMASK 2666
 RTF_BLACKHOLE 2665
 RTF_CLONING 2665
 RTF_DONE 2665
 RTF_DYNAMIC 2665
 RTF_GATEWAY 2665
 RTF_HOST 2665
 RTF_LLINFO 2665
 RTF_MASK 2665
 RTF_MODIFIED 2665
 RTF_PROTO1 2665
 RTF_PROTO2 2665
 RTF_REJECT 2665
 RTF_STATIC 2665
 RTF_UP 2665
 RTF_XRESOLVE 2665
 RTLD_DEFAULT 461
 RTLD_GLOBAL 457
 RTLD_GROUP 457
 RTLD_LAZY 456
 RTLD_LOCAL 457
 RTLD_NEXT 461
 RTLD_NODELETE 458
 RTLD_NOLOAD 458
 RTLD_NOSHARE 458
 RTLD_NOW 456
 RTLD_WORLD 457
 RTM_ADD 2665
 RTM_CHANGE 2665
 RTM_DELADDR 2665
 RTM_DELETE 2665
 RTM_GET 2665
 RTM_IFINFO 2665
 RTM_LOSING 2665
 RTM_MISS 2665
 RTM_NEWADDR 2665
 RTM_REDIRECT 2665
 RTM_RESOLVE 2665
 RTV_EXPIRE 2666
 RTV_HOPCOUNT 2666
 RTV_MTU 2666
 RTV_RPIPE 2666

RTV_RTT 2666
 RTV_RTTVAR 2666
 RTV_SPIPE 2666
 RTV_SSTHRESH 2666
 RUN_LVL 987, 3594
 runmask 2041, 2043, 2076, 2105, 3094, 3095, 3412, 3413
 rusage 945
 ruserok() 2696

S

S_IEXEC 1192, 3165
 S_IFBLK 1192, 1197, 3166
 S_IFCHR 1192, 1197, 3166
 S_IFDIR 1192, 1197, 1642, 3166
 S_IFIFO 1192, 1197, 1642, 3166
 S_IFLNK 1192, 1197, 3166
 S_IFMT 1192, 1197, 1435, 3166
 example of use 1435
 S_IFNAM 1192, 1197, 3166
 S_IFREG 1192, 1197, 3166
 S_IFSOCK 1192, 1197, 1435, 3166
 S_IREAD 1192, 3165
 S_IRGRP 3165, 3557
 S_IROTH 3165, 3557
 S_IRUSR 3165, 3557
 S_IRWXG 2814, 3165, 3557
 S_IRWXO 2814, 3165, 3557
 S_IRWXU 2814, 3165, 3557
 S_ISBLK() 779, 3167
 S_ISCHR() 779, 3167
 S_ISDIR() 780, 3167
 S_ISFIFO() 780, 3167
 S_ISGID 289, 290, 293, 612, 616, 781, 1503, 1637, 3166, 3539
 S_ISLNK() 780, 3167
 S_ISNAM() 3167
 S_ISREG() 780, 3167
 S_ISSOCK() 3167
 S_ISUID 289, 290, 293, 612, 616, 780, 1503, 3166, 3539
 S_ISVTX 289, 1637
 S_IWGRP 3165, 3557
 S_IWOTH 3165, 3557
 S_IWRITE 1192, 3165
 S_IWUSR 3165, 3557
 S_IXGRP 3165, 3557
 S_IXOTH 3165, 3557
 S_IXUSR 3165, 3557
 S_TYPEISMQ() 780, 3167
 S_TYPEISSEM() 780, 3167
 S_TYPEISSHM() 780, 3167
 SA_NOCLDSTOP 2975, 2982
 SA_ONSTACK 551
 SA_SIGINFO 1520, 2949, 2982, 3008
 sadb_x_ipsecrequest 1406
 sadb_x_policy 1406
 SAT (System Analysis Toolkit) 1171
 sbrk() 2702
 scalb() 2705
 scalbn(), scalbnf() 2707
 scandir() 2712
 scanf() 2713

SCHED_ADJTOHEAD 2780
SCHED_ADJTOTAIL 2781
SCHED_APS_ADD_SECURITY 2767
SCHED_APS_ATTACH_EVENTS 2766
SCHED_APS_BNKR_BASIC 2749
SCHED_APS_BNKR_CANCEL_BUDGET 2749
SCHED_APS_BNKR_LOG (not implemented) 2749
SCHED_APS_BNKR_REBOOT 2749
SCHED_APS_BNKR_RECOMMENDED 2749
sched_aps_clear_crit_parms 2763
SCHED_APS_CLEAR_CRITICAL 2763
sched_aps_create_parms 2751
SCHED_APS_CREATE_PARTITION 2751
sched_aps_events_parm 2766
sched_aps_info 2746
sched_aps_join_parms 2756
SCHED_APS_JOIN_PARTITION 2756
SCHED_APS_LOOKUP 2755
sched_aps_lookup_parms 2755
sched_aps_mark_crit_parms 2762
SCHED_APS_MARK_CRITICAL 2762
sched_aps_modify_parms 2758
SCHED_APS_MODIFY_PARTITION 2757
sched_aps_overall_stats 2761
SCHED_APS_OVERALL_STATS 2761
sched_aps_parms 2750
sched_aps_partition_info 2753
sched_aps_partition_stats 2759
SCHED_APS_PARTITION_STATS 2759
SCHED_APS_PSTATS_IS_BANKRUPT_NOW 2760
SCHED_APS_PSTATS_WAS_BANKRUPT 2760
SCHED_APS_QUERY_PARMS 2746
SCHED_APS_QUERY_PARTITION 2753
SCHED_APS_QUERY_PROCESS 2771
sched_aps_query_process_parms 2771
SCHED_APS_QUERY_THREAD 2764
sched_aps_query_thread_parms 2764
SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO 2748
SCHED_APS_SEC_BASIC 2769
SCHED_APS_SEC_FLEXIBLE 2769
SCHED_APS_SEC_JOIN_SELF_ONLY 2771
SCHED_APS_SEC_NONZERO_BUDGETS 2770
SCHED_APS_SEC_OFF 2768
SCHED_APS_SEC_PARENT_JOINS 2770
SCHED_APS_SEC_PARENT_MODIFIES 2770
SCHED_APS_SEC_PARTITIONS_LOCKED 2768, 2771
SCHED_APS_SEC_RECOMMENDED 2769
SCHED_APS_SEC_ROOT_JOINS 2770
SCHED_APS_SEC_ROOT_MAKES_CRITICAL 2770
SCHED_APS_SEC_ROOT_MAKES_PARTITIONS 2770
SCHED_APS_SEC_ROOT_OVERALL 2769
SCHED_APS_SEC_SYS_JOINS 2770
SCHED_APS_SEC_SYS_MAKES_CRITICAL 2770
SCHED_APS_SEC_SYS_MAKES_PARTITIONS 2770
sched_aps_security_parms 2768
SCHED_APS_SET_PARMS 2750
SCHED_FIFO 2223, 2725, 2727, 2740, 2775, 2777, 3096
sched_get_priority_adjust() 2723
sched_get_priority_max() 2725
sched_get_priority_min() 2727
sched_getparam() 2720
sched_getscheduler() 2729
SCHED_NOCHANGE 2223, 3405
SCHED_OTHER 2223, 2725, 2727, 2740, 2775, 2777, 3096
sched_param 2731
SCHED_PRIO_LIMIT_ERROR() 2402, 2732, 2890
SCHED_PRIO_LIMIT_SATURATE() 2402, 2732, 2890
SCHED_RR 2223, 2725, 2727, 2740, 2775, 2777, 3096
sched_rr_get_interval() 2736
sched_setparam() 2145, 2738
sched_setscheduler() 2145, 2741
SCHED_SPORADIC 2223, 2775, 3096
sched_yield() 2743
SchedCtl() 2142
SchedCtl(), SchedCtl_r() 2745
SchedGet() 2147
SchedGet(), SchedGet_r() 2774
SchedInfo(), SchedInfo_r() 2778
SchedSet() 2145, 2147
SchedSet(), SchedSet_r() 2781
scheduling 2041, 2042, 2078, 2080, 2107, 2109, 2195, 2197, 2199, 2219, 2221, 2223, 2299, 2400, 2720, 2729, 2731, 2741, 2774, 2778, 2781, 2783, 3094, 3403, 3405
 information, getting 2778
 parameters 2041, 2042, 2078, 2080, 2107, 2109, 2195, 2197, 2219, 2221, 2299, 2400, 2720, 2731, 2774, 2781, 3094, 3403
 getting 2720
 threads, getting for 2197, 2221, 2299, 2774
 threads, inheriting 2195, 2219, 3403
 threads, setting for 2041, 2042, 2078, 2080, 2107, 2109, 2400, 2781, 3094
 policy 2041, 2042, 2078, 2080, 2107, 2109, 2195, 2199, 2219, 2223, 2299, 2400, 2729, 2741, 2774, 2781, 3094, 3403, 3405
 don't change 2223
 FIFO 2223
 other 2223
 processes, getting for 2729
 processes, setting for 2041, 2042, 2078, 2080, 2107, 2109, 2741, 3094
 round-robin 2223
 sporadic 2223
 threads, getting for 2199, 2299, 2774
 threads, inheriting 2195, 2219, 3403
 threads, setting for 2223, 2400, 2781, 3405
 yielding 2783
SchedYield(), SchedYield_r() 2783
SCM_RIGHTS 3578
scoped addresses 1101
scripts, running 569, 592, 2043, 3090, 3094, 3100, 3106, 3112, 3117, 3123, 3130, 3135, 3141, 3146
searchenv() 2785
secure Internet Protocol, See IPsec
secure RPC domain 345
security 2767, 2768, 2843
 adaptive partitions, setting for 2767, 2768
 file descriptor, setting lowest 2843
Security Policy Database (SPD) 1399
seed48() 2787
SEEK_CUR 626, 772, 1286, 1351, 1555, 1562
SEEK_END 626, 772, 1286, 1351, 1555, 1562
SEEK_SET 625, 626, 772, 1286, 1352, 1555, 1562

seekdir() 2789
 segments 237, 491, 494, 549, 2900
 data 2900
 maximum size 2900
 data, end of 491, 494
 text 237, 549
 beginning of 237
 end of 549
 select_attach() 2797
 select_attr_t 2796
 select_detach() 2800
 SELECT_FLAG_EXCEPT 2797
 SELECT_FLAG_READ 2797
 SELECT_FLAG_REARM 2797
 SELECT_FLAG_SRVECEPT 2797
 SELECT_FLAG_WRITE 2797
 select_query() 2803
 select() 59, 2792, 3766
 sem_close() 2805
 sem_destroy() 2807
 SEM_FAILED 2816
 sem_getvalue() 2809
 sem_init() 2811
 sem_open() 2815
 sem_post() 2818
 sem_timedwait(), sem_timedwait_monotonic() 2820
 sem_trywait() 2823
 sem_unlink() 2825
 SEM_VALUE_MAX 2811, 2812, 2815
 sem_wait() 2827
 semaphores 2805, 2807, 2809, 2811, 2815, 2818, 2820,
 2823, 2825, 2827, 3289, 3301, 3303, 3305,
 3310
 maximum number of (_SC_SEM_NSEMS_MAX) 3310
 named 2805, 2815, 2825
 accessing and creating 2815
 closing 2805
 destroying 2825
 posting 2818, 3301
 unnamed 2807, 2811, 3289, 3305
 destroying 2807, 3289
 initializing 2811, 3305
 value 2809, 2811, 2815, 2818, 2820, 2823, 2827,
 3301, 3303
 decrementing 2820, 2823, 2827, 3303
 getting 2809
 incrementing 2818, 3301
 setting 2811, 2815
 waiting on 2820, 2823, 2827, 3303
 with a time limit 2820
 without blocking 2823
 send() 2830
 sendmsg() 2834
 sendto() 2838
 serial number, getting 345
 serial ports 1672, 1677, 1680, 1686
 opening 1672
 reading 1677
 script, running on 1680
 writing 1686
 servent 2842
 server attributes, getting 1211
 servers 278, 284, 304, 356, 361, 363, 372, 469, 472,
 474, 476, 478, 480, 481, 1964, 1975
 connections 278, 284, 304, 356, 361, 363, 372
 creating 278
 destroying 284
 information about 304, 356, 361, 363, 372
 data server 469, 472, 474, 476, 478, 480, 481
 applications, registering and deregistering 474, 480
 variables, creating and destroying 469, 472
 variables, flags 476
 variables, getting and setting 478, 481
 PCI 1964, 1975
 attaching 1964
 detaching 1975
 services 525, 952, 954, 956, 2842, 2905
 database 525, 952, 954, 956, 2842, 2905
 closing 525
 entries, getting 952, 954, 956
 entry structure 2842
 opening 2905
 sessions 552, 558, 564, 575, 581, 587, 597, 958, 1547,
 2043, 2156, 2159, 2164, 2176, 2637, 2889,
 2907, 3092, 3094, 3124, 3351
 character device terminal drivers, support for 2176
 controlling terminal 3351
 creating 1547, 2907
 current 2637
 disassociating 597
 ID, getting 958
 leader, creating 2043, 2889, 3094
 membership, inheriting 552, 558, 564, 575, 581, 587,
 3124
 remote node 3092
 system daemons 2156, 2159, 2164
 termination, notification of 2159, 2164
 set_lowest_fd() 2843
 setbuf() 2845
 setbuffer() 2847
 setdomainname() 2849
 setgid() 2147, 2851
 setenv() 570, 592, 2853, 3101, 3113, 3130, 3142
 seteuid() 2147, 2856
 setfsent() 2860
 setgid() 784, 2147, 2858, 3171
 not honoring on mounted filesystems 784, 3171
 setgrent() 2862
 setgroups() 2147, 2864
 sethostent() 867, 873, 2866
 sethostname() 2868
 SETIOV() 2870
 setitimer() 2872
 setjmp() 2875
 setkey() 2877
 setlinebuf() 2878
 setlocale() 2880, 3268
 setlogmask() 2882
 setnetent() 2884
 setpgid() 2886
 setpgrp() 2145, 2889
 setprio() 2145, 2890
 setprotoent() 2892
 setpwent() 2894

- setregid() 2147, 2895
- setreuid() 2147, 2897
- setrlimit() 2146
- setrlimit(), setrlimit64() 2899
- setservernt() 2905
- setsid() 2907
- setsockopt() 2910
- setspent() 2912
- setstate() 2913
- settimeofday() 2143, 2915
- setuid() 784, 1691, 2147, 2917, 3171
 - not honoring on mounted filesystems 784, 1691, 3171
- setutent() 2919
- setvbuf() 2921
- sh 2019
- SH_COMPAT 1193, 1328, 3083, 3087
- SH_DENYNO 1193, 1329, 3084, 3087
- SH_DENYRD 1193, 1329, 3084, 3087
- SH_DENYRW 1193, 1329, 3084, 3087
- SH_DENYWR 1193, 1329, 3084, 3087
- shadow password database 526, 654, 971, 975, 2459, 2912
 - closing 526
 - entries 654, 971, 975, 2459
 - reading 654, 971, 975
 - structure 2459
 - writing 2459
 - rewinding 2912
- shared locks 666
- shared memory 1656, 1699, 1832, 1838, 2925, 2931, 2935, 2940
 - access protection, changing 1699
 - attributes, modifying 2925, 2931
 - mapping 1656
 - opening 2935
 - removing 2940
 - unmapping 1832, 1838
- shared objects 449, 451, 453, 455, 459, 461
 - addresses, translating 449
 - closing 451
 - debugging 459
 - errors 453
 - opening 455
 - symbol, getting address of 461
- SHELL 300, 2019, 3330
- shell scripts, running 569, 592, 2043, 3090, 3094, 3100, 3106, 3112, 3117, 3123, 3130, 3135, 3141, 3146
- shm_ctl_special() 2144, 2931
- shm_ctl() 2144, 2925
- shm_open() 2935
- shm_unlink() 2940
- SHMCTL_ANON 2925, 2931
- SHMCTL_GLOBAL 2144, 2925, 2931
- SHMCTL_HIGHUSAGE 909, 2925, 2931
- SHMCTL_ISADMA 2925, 2931
- SHMCTL_LAZY 2925, 2931
- SHMCTL_LAZYWRITE 2925, 2927, 2931
- SHMCTL_LOWERPROT 2925, 2932
- SHMCTL_NODEFRAG 2926, 2932, 3862
- SHMCTL_NOX64K 2926, 2932
- SHMCTL_PHYS 2144, 2926, 2932
- SHMCTL_PRIV 2926, 2932
- SHMCTL_REPEAT 2926, 2932
- shutdown_classify() 2945
- shutdown_display() 2945
- shutdown_done() 2946
- shutdown_error() 2946
- shutdown_process() 2946
- shutdown_progress() 2946
- shutdown_prompt() 2946
- SHUTDOWN_REBOOT 2944
- SHUTDOWN_SYSTEM 2944
- shutdown_system() 2944
- shutdown() 2942
- SI_ASYNCIO 2951, 2983, 2989
- SI_MAXAVAIL 2963
- SI_MESGQ 2951, 2983, 2989
- SI_MINAVAIL 2963
- SI_QUEUE 2950, 2983, 2989
- SI_TIMER 2950, 2983, 2989
- SI_USER 2950, 2983, 2989
- side channels 351
- SIG_APPID 2986
- SIG_BLOCK 3004
- SIG_DFL 551, 2949, 2974
- SIG_ERR 2976
- SIG_IGN 551, 2950, 2974, 2982, 3095
- SIG_INTR 2749
- SIG_SETMASK 3004
- SIG_UNBLOCK 3004
- SIGABRT 54, 2979
- sigaction 2981
- sigaction() 2147, 2948
- sigaddset() 2953
- SIGALRM 144, 890, 2873, 2980, 3428, 3449, 3452, 3550
 - process, sending to 3428, 3452, 3550
- sigblock() 2955
- SIGBUS 1660, 2980
- SIGCHLD 597, 945, 2041, 2949, 2950, 2975, 2976, 2980, 2982, 3664
 - default actions 2949, 2982
 - ignoring 2950, 2982
- SIGCONT 597, 2950, 2955, 2981
 - default actions 2950
- SIGDEADLK 2979, 3292
- sigdelset() 2957
- sigemptyset() 2959
- SIGEMT 2979
- SIGEV_CLEAR_CRITICAL() 2965
- SIGEV_GET_TYPE() 2961
- SIGEV_INTR 1151, 1155, 1183, 1737, 2961
- SIGEV_INTR_INIT() 2962
- SIGEV_MAKE_CRITICAL() 2965
- SIGEV_NONE 1520, 2766, 2961
- SIGEV_NONE_INIT() 2766, 2962
- SIGEV_PULSE 1150, 1155, 1708, 1737, 2961, 3428
- SIGEV_PULSE_INIT() 2962
- SIGEV_PULSE_PRIO_INHERIT 2962
- SIGEV_SIGNAL 1151, 1155, 1520, 1708, 1737, 2961, 3428
- SIGEV_SIGNAL_CODE 1151, 1155, 1708, 2961, 3428
- SIGEV_SIGNAL_CODE_INIT() 2963
- SIGEV_SIGNAL_INIT() 2963
- SIGEV_SIGNAL_THREAD 1151, 1155, 1708, 2961, 3428
- SIGEV_SIGNAL_THREAD_INIT() 2964
- SIGEV_SIGNAL_VALUE_INIT() 2963

SIGEV_THREAD 2961
 SIGEV_THREAD_INIT() 2964
 SIGEV_UNBLOCK 1737, 2961
 SIGEV_UNBLOCK_INIT() 2964
 sigevent 1388, 1736, 2182, 2749, 2762, 2766, 2961
 bankruptcy 2749, 2766
 critical 2749, 2762
 system overload 2766
 triggering 2182
 type, determining with SIGEV_GET_TYPE() 2961
 sigfillset() 2966
 SIGFPE 2979
 SIGHUP 597, 2979
 process groups, targeting 597
 SIGILL 247, 254, 2979, 3409, 3410
 siginfo_t 2950, 2983, 2988
 SIGINT 2979, 3367
 process groups, sending to 3367
 SIGIO 2949, 2980, 2982
 default actions 2949, 2982
 SIGIOT 2979
 sigismember() 2968
 SIGKILL 2408, 2951, 2955, 2975, 2979
 siglongjmp() 2970
 sigmask() 2972
 sign, copying 375
 signal() 2147, 2974
 SignalAction() 2147
 SignalAction(), SignalAction_r() 2978
 SignalKill() 2148
 SignalKill(), SignalKill_r() 2987
 SignalProcmask() 2148
 SignalProcmask(), SignalProcmask_r() 2992
 signals 50, 54, 144, 397, 597, 712, 714, 890, 945, 965,
 1155, 1494, 1496, 1660, 1685, 1687, 1832,
 1836, 1838, 1947, 2041, 2042, 2074, 2082,
 2084, 2086, 2103, 2111, 2113, 2115, 2217,
 2310, 2407, 2408, 2467, 2469, 2472, 2482,
 2873, 2900, 2948, 2949, 2950, 2951, 2953,
 2955, 2957, 2959, 2966, 2968, 2970, 2972,
 2974, 2975, 2976, 2978, 2980, 2981, 2983,
 2987, 2992, 2995, 2997, 3000, 3002, 3004,
 3007, 3010, 3012, 3014, 3016, 3018, 3020,
 3022, 3094, 3247, 3310
 actions 2041, 2042, 2074, 2082, 2084, 2086, 2103,
 2111, 2113, 2115, 2948, 2949, 2974, 2978
 default 2949
 inheriting 2041, 2042, 2074, 2082, 2084, 2086,
 2103, 2111, 2113, 2115
 blocking 712, 714, 2408, 2467, 2469, 2955, 3012
 SIGCONT 2955
 SIGKILL 2408, 2955
 SIGSTOP 2408, 2955
 SIGTTOU 712, 714, 2467, 2469
 catching 2975
 SIGKILL 2975
 SIGSTOP 2975
 floating-point not safe to use in handlers 50
 ignoring 712, 714, 945, 2467, 2469, 2950, 2951,
 2975, 2976
 SIGCHLD 945, 2976
 SIGKILL 2951, 2975
 signals (*continued*)
 ignoring (*continued*)
 SIGSTOP 2951, 2975
 SIGTTOU 712, 714, 2467, 2469
 information about 3022
 masks 2407, 2970, 2972, 2992, 3010
 constructing 2972
 restoring 2970
 saving 3010
 signal-blocked 2992
 threads 2407
 names 2978
 POSIX 2978
 process groups, targeting 597, 1494, 1496, 2951, 2983,
 2987
 processes 2949, 2995, 3002, 3007, 3310
 maximum outstanding (_SC_SIGQUEUE_MAX) 3310
 pending 3002
 queuing 2949, 3007
 suspending until delivered 2995
 QNX Neutrino 2981
 sending 54, 144, 397, 597, 712, 714, 890, 965, 1155,
 1494, 1496, 1660, 1685, 1687, 1832, 1836,
 1838, 2217, 2467, 2469, 2472, 2482, 2873,
 2900, 2949, 2975, 2987, 3094
 SIGABRT 54
 SIGALRM 144, 890, 2873
 SIGBUS 1660
 SIGCHLD 597, 2949, 2975
 SIGCONT 597
 SIGHUP 597
 SIGKILL 2975
 SIGPIPE 712, 714, 965, 1685, 1687, 2467, 2469,
 2472
 SIGSEGV 1155, 1832, 1836, 1838, 2217, 2900
 SIGSTOP 3094
 SIGTRAP 397
 SIGXCPU 2900
 SIGXFSZ 2900
 sets 2953, 2957, 2959, 2966, 2968
 adding to 2953
 initializing 2959, 2966
 membership, checking for 2968
 removing from 2957
 string describing 3247
 threads 3004, 3014
 mask 3004, 3014
 threads, targeting 2310, 2951, 2983, 2987
 unblocking 3018
 user-defined 2980
 waiting for 1947, 2997, 3000, 3014, 3016, 3020, 3022
 SignalSuspend(), SignalSuspend_r() 2995
 SignalWaitinfo(), SignalWaitinfo_r() 2997
 sigpause() 3000
 sigpending() 3002
 SIGPHOTON 2981
 SIGPIPE 712, 714, 965, 1685, 1687, 2467, 2469, 2472,
 2980
 SIGPOLL 2980
 sigprocmask() 2148, 3004
 SIGPWR 2980
 sigqueue() 2148, 3007

- SIGQUIT 2979
- SIGRTMAX 2981
- SIGRTMIN 2981
- SIGSEGV 1155, 1179, 1182, 1832, 1836, 1838, 2217, 2900, 2980, 3409, 3410
- SIGSELECT 2981
- sigsetjmp() 3010
- sigsetmask() 3012
- SIGSTOP 2408, 2950, 2951, 2955, 2975, 2980, 3094
 - default actions 2950
- sigsuspend() 3014
- SIGSYS 2980
- SIGTERM 2980
- sigtimedwait() 3016
- SIGTRAP 397, 2979
- SIGTSTP 2981
- SIGTTIN 2981, 3615
- SIGTTOU 712, 714, 2467, 2469, 2981, 3615
- sigunblock() 3018
- SIGURG 2949, 2980, 2982, 3076
 - default actions 2949, 2982
 - process groups, sending to 3076
- SIGUSR1 2980
- SIGUSR2 2980
- sigval 2964
- sigwait() 3020
- sigwaitinfo() 3022
- SIGWINCH 2949, 2980, 2982
 - default actions 2949, 2982
- SIGXCPU 2900
- SIGXFSZ 2900, 3539
- sin(), sinf(), sinl() 3024
- sines 159, 3024, 3026
 - hyperbolic 3026
 - inverse hyperbolic 159
- sinh(), sinhf(), sinhl() 3026
- SIOCADDMULTI 1199
- SIOCADDRT 1199
- SIOCAIFADDR 1199
- SIOCALIFADDR 1199
- SIOCATMARK 1199
- SIOCDAEP 1199
- SIOCDELMULTI 1199
- SIOCDELRT 1199
- SIOCDFADDR 1199
- SIOCDFPHYADDR 1199
- SIOCFLIFADDR 1199
- SIOCG80211 1207
- SIOCG80211NWID 1207
- SIOCG80211NWKEY 1207
- SIOCG80211STATS, SIOCG80211ZSTATS 1207
- SIOCGARP 1199
- SIOCGDRVSPEC 1199
- SIOCGGETSGCNT 1199
- SIOCGGETVIFCNT 1199
- SIOCGGETVLAN 1207
- SIOCGHIWAT 1199
- SIOCGIFADDR 1199
- SIOCGIFALIAS 1199
- SIOCGIFASYNCMAP 1199
- SIOCGIFBRDADDR 1199
- SIOCGIFCAP 1199
- SIOCGIFCONF 1199, 1393
- SIOCGIFDLT 1199
- SIOCGIFDSTADDR 1199
- SIOCGIFFLAGS 1199, 1393
- SIOCGIFGENERIC 1199
- SIOCGIFMEDIA 1199, 1207
- SIOCGIFMETRIC 1199
- SIOCGIFMTU 1199
- SIOCGIFNETMASK 1199
- SIOCGIFPDSTADDR 1199
- SIOCGIFSRCADDR 1199
- SIOCGLIFADDR 1199
- SIOCGLIFPHYADDR 1199
- SIOCGLOWAT 1199
- SIOCGPGRP 1199
- SIOCIFCREATE 1199
- SIOCIFDESTROY 1199
- SIOCIFGCLONERS 1199, 1208
- SIOCS80211 1208
- SIOCS80211NWID 1208
- SIOCS80211NWKEY 1208
- SIOCSARP 1199
- SIOCSDRVSPEC 1199
- SIOCSETVLAN 1208
- SIOCSHIWAT 1199
- SIOCSIFADDR 1199
- SIOCSIFASYNCMAP 1199
- SIOCSIFBRDADDR 1199
- SIOCSIFCAP 1199
- SIOCSIFDSTADDR 1199
- SIOCSIFFLAGS 1199
- SIOCSIFGENERIC 1199
- SIOCSIFMEDIA 1199
- SIOCSIFMETRIC 1199
- SIOCSIFMTU 1199
- SIOCSIFNETMASK 1199
- SIOCSIFPHYADDR 1199
- SIOCSLIFPHYADDR 1199
- SIOCSLOWAT 1199
- SIOCSPPGRP 1199
- site-local addresses 1101
- sleep() 3028
- sleeping 401, 1867, 1868, 3586
 - for microseconds 3586
 - for milliseconds 401, 1867, 1868
- sleepon locks 2409, 2411, 2413, 2415, 2418, 2419, 3030, 3032, 3034, 3036, 3038, 3040, 3042
 - destroying 3032
 - initializing 3034
 - locking 2411, 3036
 - unblocking 2409, 2413, 3030, 3038
 - unlocking 2418, 3040
 - waiting 2415, 2419, 3042
- slog2_buffer_config_t 3052
- slog2_buffer_set_config_t 3053
- SLOG2_CRITICAL 3053
- SLOG2_DEBUG1 3053
- SLOG2_DEBUG2 3053
- SLOG2_ERROR 3053
- SLOG2_FA_CHAR() 3049
- SLOG2_FA_DOUBLE() 3049
- SLOG2_FA_END 3049

SLOG2_FA_FLOAT() 3049
 SLOG2_FA_SIGNED() 3049
 SLOG2_FA_STRING() 3049
 SLOG2_FA_UNSIGNED() 3049
 slog2_get_verbosity() 3051
 SLOG2_INFO 3053
 SLOG2_MAX_BUFFERS 3052, 3053
 SLOG2_NOTICE 3053
 slog2_register() 3052
 slog2_reset() 3056
 slog2_set_default_buffer() 3057
 slog2_set_verbosity() 3059
 SLOG2_SHUTDOWN 3053
 SLOG2_WARNING 3053
 slog2c() 3045
 slog2f() 3047
 slog2fa() 3049
 slogb() 3061
 slogf() 3063
 slogi() 3066
 snprintf() 3070
 SO_BINDTODEVICE 964
 SO_BROADCAST 964
 SO_DEBUG 965
 SO_DONTROUTE 965
 SO_ERROR 965
 SO_KEEPAIVE 965
 SO_LINGER 962, 965, 2909
 SO_OOBINLINE 966
 SO_RCVBUF 966
 SO_RCVLOWAT 966
 SO_RCVTIMEO 962, 966, 2909
 SO_REUSEADDR 967
 SO_REUSEPORT 967
 SO_SNDBUF 966
 SO_SNDLOWAT 967
 SO_SNDTIMEO 962, 967, 2909
 SO_TIMESTAMP 968
 SO_TYPE 968
 SO_USELOOPBACK 968, 2665
 SOCK_DGRAM 348, 968, 1100, 1391, 1414, 3075, 3076, 3553, 3577, 3765
 SOCK_RAW 1045, 1047, 1391, 1396, 1412, 3075, 3076
 SOCK_STREAM 58, 348, 1100, 1391, 1414, 1523, 2480, 2492, 3075, 3076, 3359, 3577
 sockaddr_un 3577
 sockatmark() 3073
 sockcred 3578
 SOCKCREDSIZE() 3578
 socket() 1045, 1047, 1100, 1391, 1396, 1412, 2663, 3076, 3359, 3553, 3577
 socketpair() 3078
 sockets 58, 122, 208, 211, 348, 719, 814, 818, 960, 963, 965, 968, 1435, 1523, 1869, 1871, 2480, 2489, 2494, 2533, 2534, 2536, 2537, 2539, 2540, 2650, 2656, 2670, 2830, 2834, 2838, 2910, 2942, 3073, 3075, 3076, 3078, 3312, 3553
 addresses 122, 719, 814, 818
 errors 814
 freeing 719
 getting 818
 structure 122
 sockets (*continued*)
 connections 58, 348, 1523, 1869, 1871, 2480, 2494, 2656, 2942
 accepting on 58, 2480
 initiating 348, 1869, 2494
 listening for 1523, 2656
 shutting down 2942
 status 1871
 creating 3076, 3078
 a pair of 3078
 connected 3078
 datagrams 3075, 3076, 3553
 debugging 965
 file descriptors, testing for association 1435
 manager, getting and setting information about 3312
 messages 2533, 2534, 2536, 2537, 2539, 2540, 2830, 2834, 2838
 peeking at 2533, 2536, 2539
 receiving from 2534, 2537, 2540
 sending to 2830, 2834, 2838
 names 208, 960, 2489, 2650
 binding to 208, 2489
 getting 960, 2650
 options 963, 2910
 getting 963
 setting 2910
 out-of-band (OOB) mark 3073
 privileged IP port, binding to 211
 privileged ports, getting 2670
 raw 3075, 3076
 stream 3075, 3076
 types 965, 968, 3075
 determining 965, 968
 SOCKS 49, 2480, 2489, 2494, 2650, 2656, 2668, 3081, 3764, 3765, 3766
 classification 49
 commands, executing remotely 2668
 compiling for 3765
 initializing 3081
 library 3766
 sockets 2480, 2489, 2494, 2650, 2656
 connections 2480, 2494, 2656
 names 2489, 2650
 SOCKSinit() 3081, 3765
 SOL_SOCKET 962, 2665
 sopen() 3084
 sopenfd() 3088
 sorting 149, 2476, 2712
 directory entries 149, 2712
 quick sort 2476
 space, amount free for a filesystem 784, 3171
 space, testing a character for 1451, 1475
 spawn attributes 2041, 2059, 2061, 2064, 2066, 2068, 2070, 2072, 2074, 2076, 2078, 2080, 2082, 2084, 2086, 2088, 2090, 2093, 2095, 2098, 2101, 2103, 2105, 2107, 2109, 2111, 2113, 2115, 2117, 2119
 adaptive partitions 2059, 2061, 2072
 destroying 2064
 flags 2068, 2090, 2098, 2119
 POSIX 2068, 2090, 2098, 2119
 QNX Neutrino 2090, 2119

- spawn attributes (*continued*)
 - group ID 2066, 2095
 - initializing 2093
 - process group 2074, 2103
 - remote node 2070, 2101
 - runmask 2076, 2105
 - scheduling parameters 2078, 2107
 - scheduling policy 2080, 2109
 - signals 2082, 2084, 2086, 2111, 2113, 2115
 - default actions 2082, 2111
 - ignored 2084, 2113
 - masked 2086, 2115
 - stack size, maximum 2088, 2117
 - user ID 2066, 2095
- spawn file actions 2048, 2050, 2052, 2055, 2057
 - close 2048
 - destroying 2055
 - duplicate 2050
 - initializing 2057
 - open 2052
- SPAWN_ALIGN_DEFAULT 3094
- SPAWN_ALIGN_FAULT 3094
- SPAWN_ALIGN_MASK 3095
- SPAWN_ALIGN_NOFAULT 3094
- SPAWN_CHECK_SCRIPT 3094
- SPAWN_DEBUG 3094
- SPAWN_EXEC 3094
- SPAWN_EXPLICIT_CPU 3094, 3095
- SPAWN_EXPLICIT_SCHED 3094, 3096
- SPAWN_FDCLOSED 3089, 3093, 3122
- SPAWN_HOLD 3094
- SPAWN_NEWAPP 3094
- SPAWN_NEWPGROUP 3095
- SPAWN_NOZOMBIE 3094
- SPAWN_PADDR64_SAFE 3094
- SPAWN_SEARCH_PATH 3094
- SPAWN_SETGROUP 3091, 3094, 3095, 3124
- SPAWN_SETND 3094
- SPAWN_SETSID 3094
- SPAWN_SETSIGDEF 3092, 3095, 3125
- SPAWN_SETSIGIGN 3095
- SPAWN_SETSIGMASK 3091, 3095, 3125
- SPAWN_SETSTACKMAX 3095, 3096
- SPAWN_TCSETPGROUP 3095
- spawn() 2142, 3089
- spawn* family of functions 1913
- spawnl() 3100
- spawnle() 3106
- spawnlp() 3112
- spawnlpe() 3117
- spawnp() 3122
- spawnv() 3130
- spawnve() 3135
- spawnvp() 3141
- spawnvpe() 3146
- SPD (Security Policy Database) 1399
- special characters 1686
- spinlocks 2422, 2424, 2426, 2428, 2430
 - destroying 2422
 - initializing 2424
 - locking 2426, 2428
 - unlocking 2430
- sporadic scheduling 2223
- sprintf() 3151
- spwd 2459
- sqrt(), sqrtf(), sqrtl() 3153
- square roots 3153
- srand() 3155
- srand48() 3157
- srandom() 3158
- SS_REPL_MAX 2733
- sscanf() 3162
- st_atime 1637, 1639, 2011, 2503, 2525
- st_ctime 293, 613, 616, 1503, 1516, 1637, 1639, 2011, 2658, 3739
- st_mtime 1639, 2011
- st_mode 3166
- st_mtime 1516, 1637, 1639, 2011, 2658, 3739
- ST_NOATIME 784, 3171
- ST_NOCREAT 784, 3171
- ST_NOEXEC 784, 3171
- ST_NOSUID 551, 784, 3092, 3171
- ST_OFF32 784, 3171
- ST_RDONLY 784, 3171
- stack 147, 2041, 2043, 2088, 2117, 2203, 2205, 2207, 2209, 2211, 2217, 2227, 2229, 2231, 2233, 2235, 2900, 3095, 3310
 - maximum size 2041, 2043, 2088, 2117, 2900, 3095
 - memory, allocating from 147
 - overflow, protecting against 2217
 - preallocating 2209, 2233
 - threads 2203, 2205, 2207, 2211, 2227, 2229, 2231, 2235, 3310
 - minimum size (_SC_THREAD_STACK_MIN) 3310
- stat 779, 3164
- stat(), stat64() 1353, 1355, 1558, 3164
 - resource managers, implementing in 1353, 1355
- STATE_CONDVAR 3284, 3465
- STATE_INTR 3465
- STATE_JOIN 3465
- STATE_MUTEX 3283, 3284, 3465
- STATE_NET_SEND 1791
- STATE_RECEIVE 3465
- STATE_REPLY 1782, 3465
- STATE_SEM 3465
- STATE_SEND 1782, 3465
- STATE_SIGSUSPEND 3465
- STATE_SIGWAITINFO 3465
- static rules filter, controlling 3515
- statvfs, statvfs64 783, 3170
- statvfs(), statvfs64() 3170
- stderr 536, 621, 661, 904, 1017, 2009, 2156, 2848, 2878, 3175, 3612, 3645, 3669
 - buffering 2848, 2878
 - command-line options, errors when parsing 904
 - daemons 2156
 - formatted messages on 536, 3612, 3645, 3669
 - host errors 1017
- STDERR_FILENO 661, 3175
- stderrchild process, specifying for 3089, 3122
- stdin 621, 648, 661, 827, 829, 950, 995, 2156, 2713, 3176, 3626, 3649, 3748
 - characters, reading 648, 827, 829
 - daemons 2156

stdin (continued)

- input, formatted 2713, 3626, 3649, 3748
- strings, reading 950
- wide characters, reading 995
- STDIN_FILENO 661, 2011, 2020, 3176
- stdinchild process, specifying for 3089, 3122
- STDIO_DEFAULT_BUFSIZE 2921
- stdout 621, 661, 707, 2133, 2156, 2451, 2453, 2457, 2468, 2848, 2878, 3177, 3624, 3647, 3736
 - buffering 2848, 2878
 - characters, writing 707, 2451, 2453
 - daemons 2156
 - output, formatted 2133, 3624, 3647, 3736
 - strings, writing 2457
 - wide characters, writing 2468
- STDOUT_FILENO 661, 2011, 2020, 3177
- stdoutchild process, specifying for 3089, 3122
- storage, synchronizing memory with 1825
- straddstr() 3178
- strcasecmp() 3180
- strcat() 3182
- strchr() 3184
- strcmp() 3186
- strcmpi() 3188
- strcoll() 3190, 3268
- strcpy() 3192
- strcspn() 3194
- strdup() 3196
- stream I/O 302, 618, 621, 635, 637, 639, 641, 645, 648, 650, 652, 656, 658, 661, 669, 673, 681, 702, 704, 707, 709, 711, 713, 715, 721, 770, 772, 776, 789, 796, 801, 804, 806, 808, 810, 823, 825, 827, 829, 950, 991, 993, 995, 1143, 1645, 1647, 2007, 2009, 2011, 2019, 2133, 2447, 2449, 2451, 2453, 2457, 2464, 2466, 2468, 2646, 2713, 2845, 2847, 2878, 2921, 3478, 3573, 3575, 3616, 3618, 3620, 3622, 3624, 3626, 3647, 3649, 3736, 3748
 - buffering 2845, 2847, 2878, 2921
 - associating with 2845, 2921
 - block 2847
 - line 2878
 - characters 645, 648, 652, 704, 707, 823, 825, 827, 829, 993, 995, 2447, 2449, 2451, 2453, 2466, 2468, 3573, 3575
 - pushing back 3573, 3575
 - reading 645, 648, 652, 823, 825, 827, 829, 993, 995
 - writing 704, 707, 2447, 2449, 2451, 2453, 2466, 2468
 - closing 618, 621
 - end-of-file 302, 637
 - clearing 302
 - errors 302, 639, 2009
 - clearing 302
 - messages, printing 2009
 - file descriptors 635, 661
 - associating streams with 635
 - getting 661
 - files 641, 669, 673, 681, 796, 801, 1143
 - flushing 641, 673
 - locking 669, 796

stream I/O (continued)

- files (*continued*)
 - opening 681
 - reading 1143
 - unlocking 801
- input, formatted 770, 810, 2713, 3618, 3622, 3626, 3649, 3748
- output, formatted 702, 806, 2133, 3616, 3620, 3624, 3647, 3736
- pipes 2007, 2011, 2019
 - closing 2007
 - creating 2011
 - opening 2019
- position 650, 772, 776, 789
 - getting 650, 789
 - setting 772, 776
- reading 715
- reopening 721
- rewinding 2646
- seeking 772
- strings 709, 713, 950, 2457
 - reading 950
 - writing 709, 713, 2457
- telling 789
- temporary files 1645, 1647, 3478
- wide characters 656, 658, 711, 804
 - orientation 804
 - reading 656, 658
 - writing 711
- words 991, 2464
 - getting next 991
 - writing 2464
- writing 808
- stream sockets 3075, 3076
- stream, returning to remote command 2491, 2668
- streams 3345
 - flushing 3345
- strerror_r() 3200
- strerror() 3198
- strftime() 3202
- stricmp() 3208
- strings 41, 155, 171, 173, 175, 177, 202, 204, 206, 239, 344, 385, 389, 424, 492, 652, 678, 702, 709, 713, 770, 950, 1071, 1073, 1075, 1087, 1089, 1091, 1094, 1096, 1483, 1560, 1605, 1609, 1616, 1877, 1883, 2009, 2133, 2457, 2474, 2531, 2652, 2713, 2877, 3070, 3151, 3162, 3178, 3180, 3182, 3184, 3186, 3188, 3190, 3192, 3194, 3196, 3198, 3200, 3202, 3208, 3210, 3213, 3215, 3217, 3219, 3221, 3223, 3225, 3227, 3229, 3231, 3233, 3239, 3241, 3243, 3245, 3247, 3249, 3251, 3253, 3257, 3258, 3260, 3262, 3264, 3266, 3268, 3555, 3598, 3616, 3618, 3624, 3626, 3633, 3635, 3637, 3641, 3674, 3682, 3684, 3693, 3695, 3699
 - See also characters; wide characters
 - character, filling with 3229, 3245
 - comparing 204, 1609, 1616, 3180, 3186, 3188, 3190, 3208, 3217, 3221, 3225
 - case-insensitive 3180, 3188, 3208, 3217, 3225
 - case-sensitive 3186, 3221

- strings (*continued*)
 - comparing (*continued*)
 - locale's collating sequence, using 3190
 - concatenating 3178, 3182, 3210, 3219
 - configuration 344
 - getting 344
 - setting 344
 - copying 206, 1605, 3192, 3196, 3210, 3223, 3268
 - encrypting 385, 492, 2474, 2877
 - error messages 2009, 3198, 3200
 - formatted 3070, 3151, 3633, 3635
 - hexadecimal numbers, converting to/from 173
 - input, formatted 770, 2713, 3618, 3626
 - IP addresses, converting to/from 1094, 1096
 - IPv4 addresses, converting to/from 1073, 1075, 1089, 1091
 - length 3213, 3227
 - lowercase, converting to 3215
 - matching 678
 - network numbers, converting to/from 1087
 - node descriptors, converting to/from 1877, 1883
 - numbers, converting to/from 171, 173, 175, 177, 1483, 1560, 3253, 3257, 3262, 3264, 3555, 3598
 - output, formatted 702, 2133, 3616, 3624
 - paths, resolving 2531
 - reversing 3241
 - scanning input from 3162, 3637, 3641
 - searching 202, 424, 1071, 2652, 3184, 3194, 3231, 3239, 3249, 3251, 3674, 3682, 3693, 3695, 3699
 - characters 1071, 2652, 3184, 3239
 - sets of characters 3194, 3231, 3249
 - sets of wide characters 3682, 3693, 3699
 - slashes (/) 202, 424
 - strings 3251
 - wide characters 3674, 3695
 - signal descriptions 3247
 - splitting 3243, 3258, 3260
 - stdin, reading from 950
 - stdout, writing to 2457
 - streams 652, 709, 713
 - reading from 652
 - writing 709, 713
 - substrings 3217, 3221, 3225
 - comparing, case-insensitive 3217, 3225
 - comparing, case-sensitive 3221
 - time_t, converting to/from 389
 - time, formatted 3202, 3233, 3684
 - tm, converting to/from 155
 - tokens, splitting into 3258, 3260
 - transforming 3268
 - uppercase, converting to 3266
 - zeroing 239
 - See also characters; wide characters
- strlcat() 3210
- strncpy() 3210
- strlen() 3213
- strlwr() 3215
- strncasecmp() 3217
- strncat() 3219
- strncmp() 3221, 3268
- strncpy() 3223, 3268
- strnicmp() 3225
- strnlen() 3227
- strnset() 3229
- strpbrk() 3231
- strptime() 3233
- strrchr() 3239
- strrev() 3241
- strsep() 3243
- strset() 3245
- strsignal() 3247
- strspn() 3249
- strstr() 3251
- strtod() 3253
- strtof() 3253
- strtoimax() 3257
- strtok_r() 3260
- strtok() 3258
- strtol(), strtoll() 3262
- strtold() 3253
- strtoul(), strtoull() 3264
- strtoumax() 3257
- struct inotify_event 1139
- struct, offset of members within 1900
- strupr() 3266
- strxfrm() 3268
- suboptions, parsing 977
- SUN_LEN() 3577
- swab() 3270
- swprintf() 3272
- swscanf() 3274
- symbolic links 1502, 1558, 1943, 1945, 2520, 2531, 2556, 3276, 3579
 - creating 3276
 - deleting 2556, 3579
 - information, getting 1558
 - ownership, changing 1502
 - reading 2520
 - resolving 2531
 - temporary 1943, 1945
- symlink() 3276
- SYMLOOP_MAX 2521
- sync_t 3305
- sync() 1360, 1362, 3279
 - resource managers, implementing in 1360, 1362
- SyncCondvarSignal(), SyncCondvarSignal_r() 3280
- SyncCondvarWait(), SyncCondvarWait_r() 3282
- SyncCtl() 2145
- SyncCtl(), SyncCtl_r() 3286
- SyncDestroy(), SyncDestroy_r() 3289
- synchronization objects 41, 3280, 3282, 3286, 3289, 3291, 3294, 3297, 3299, 3301, 3303, 3305
 - See also mutexes; semaphores; threads
 - creating 3305
 - destroying 3289
 - mutexes 3286, 3291, 3294, 3297, 3299
 - events 3286, 3291
 - locking 3294, 3299
 - priority 3286
 - reviving 3297
 - semaphores 3301, 3303
 - incrementing 3301, 3303

synchronization objects (*continued*)

- threads 3280, 3282
 - blocking 3282
 - waking up 3280
 See also mutexes; semaphores; threads
- synchronous I/O, support for (_PC_ASYNC_IO) 700, 1937
- SyncMutexEvent() 2143
- SyncMutexEvent(), SyncMutexEvent_r() 3291
- SyncMutexLock(), SyncMutexLock_r() 3294
- SyncMutexRevive(), SyncMutexRevive_r() 3297
- SyncMutexUnlock(), SyncMutexUnlock_r() 3299
- SyncSemPost(), SyncSemPost_r() 3301
- SyncSemWait(), SyncSemWait_r() 3303
- SyncTypeCreate() 2145
- SyncTypeCreate(), SyncTypeCreate_r() 3305
- sys_errlist 539
- sys_nerr 540
- sys_nsig 2978
- sys_siglist 2978
- sysconf() 3308
- sysctl() 3312
- syslog() 3316
- sysmgr_reboot() 2146, 3319
- sysmgr_runstate_burst() 2146, 3322
- sysmgr_runstate(), sysmgr_runstate_dynamic() 2146, 3320
- SYSPAGE_CPU_ENTRY() 3324
- SYSPAGE_ENTRY() 3326, 3327
 - qtime 3327
 - boot_time 3327
 - cycles_per_sec 3327
- system 323, 329, 333, 345, 394, 835, 943, 945, 1028, 1031, 1032, 1035, 1037, 1039, 2156, 2158, 2159, 2163, 2164, 2168, 2170, 2178, 2674, 2681, 2684, 2686, 2693, 2749, 2899, 2944, 3308, 3319, 3327, 3330
 - clock 329, 333, 3327
 - getting 333
 - period, getting and setting 329
 - setting 333
 - ticks per second 3327
 - commands, executing 3330
 - daemons 394, 2156, 2159, 2164
 - termination, notification of 2159, 2164
 - events 2158, 2163, 2168, 2170, 2178
 - notification of 2158, 2163, 2168, 2178
 - triggering 2170
 - hardware information 1028, 1031, 1032, 1035, 1037, 1039
 - instruction set architecture 345
 - limits 3308
 - getting 3308
 - rebooting 3319
 - rebooting on bankruptcy 2749
 - resources 835, 943, 945, 2674, 2681, 2684, 2686, 2693, 2899
 - creating 2681
 - destroying 2684
 - limits, getting 835, 943
 - limits, setting 2899
 - querying 2693
 - reserving 2674
 - returning 2686

system (*continued*)

- resources (*continued*)
 - usage, getting 945
 - shutting down 2944
 - time since booting 3327
 - time, adjusting 323
- System Analysis Toolkit (SAT) 1171
- system databases 385, 495, 496, 524, 526, 654, 842, 844, 847, 851, 853, 855, 857, 859, 930, 932, 934, 937, 938, 941, 971, 975, 1131, 2459, 2474, 2860, 2862, 2894, 2912
- filesystems 495, 842, 844, 847, 2860
 - closing 495
 - getting next entry 844
 - opening 2860
 - searching for a block special device 847
 - searching for a filesystem 842
- groups 496, 851, 853, 855, 857, 859, 1131, 2862
 - closing 496
 - ID, getting information about 853, 855
 - membership 1131
 - name, getting information about 857, 859
 - next entry, getting 851
 - rewinding 2862
- passwords 385, 524, 930, 932, 934, 937, 938, 941, 2474, 2894
 - closing 524
 - encrypting 385, 2474
 - entry, getting for a user 934, 937, 938, 941
 - entry, getting next 930, 932
 - rewinding 2894
- shadow passwords 526, 654, 971, 975, 2459, 2912
 - closing 526
 - entry, reading 654, 971, 975
 - entry, structure 2459
 - entry, writing 2459
 - rewinding 2912
- system message log 340, 1920, 2882, 3061, 3063, 3066, 3316, 3631, 3643
 - closing 340
 - log priority mask 2882
 - opening 1920
 - writing to 3061, 3063, 3066, 3316, 3631, 3643
 - blocks 3061
 - formatted output 3063, 3631
 - integers 3066
- system message log (slogger2) 3045, 3047, 3049, 3051, 3052, 3056, 3057, 3059, 3629
 - default buffer, setting 3057
 - logging a constant string 3045
 - logging a formatted string 3047, 3049, 3629
 - registering with 3052
 - resetting 3056
 - verbosity 3051, 3059
- system packet forwarding database, See ROUTE
- system page 3324, 3326, 3329
 - CPU-specific entry, getting a pointer to 3324
 - entry, getting a pointer to 3326
- System partition 2751
- system() 3330

T

- T_PTR 2566, 2569, 2572, 2575
- tan(), tanf(), tanl() 3334
- tangents 167, 3334, 3336
 - hyperbolic 3336
 - inverse hyperbolic 167
- tanh(), tanhf(), tanhl() 3336
- target operating system 51
- tcdrain() 3338
- tcdropline() 3340
- tcflow() 3342
- TCFLSH 1199
- tcflush() 3345
- TCGETA 1199
- tcgetattr() 3347
- tcgetpgrp() 3349
- TCGETS 1199
- tcgetsid() 3351
- tcgetsize() 3353
- TCIFLUSH 3345
- tcinject() 3355
- TCIOFF 3343
- TCIOFFHW 3343
- TCIOFLUSH 3345
- TCION 3343
- TCIONHW 3343
- tcischars() 3357
- TCOFLUSH 3345
- TCOOFF 3342
- TCOOFFHW 3342
- TCOON 3343
- TCOONHW 3343
- TCP (Transmission Control Protocol) 497, 3359, 3766
 - connection, closing 497
 - SOCKS 3766
- TCP_KEEPALIVE 3360
- TCP_MAXSEG 3360
- TCP_NODELAY 968, 3360
- TCP/IP 122, 211, 465, 466, 522, 523, 525, 872, 876, 878, 881, 1012, 1017, 1024, 1073, 1075, 1078, 1083, 1089, 1091, 1094, 1096, 2534, 2537, 2540, 2830, 2834, 2838, 2842, 2866, 2884, 2892, 2905
 - address information 122
 - addrinfo 122
 - addresses 1073, 1075, 1078, 1083, 1089, 1091, 1094, 1096
 - manipulating 1078
 - network numbers, extracting 1083
 - strings, converting to/from 1073, 1075, 1089, 1091, 1094, 1096
 - errors 1012, 1017, 1024
 - host entries 872, 876, 878, 881
 - getting 872, 876, 878, 881
 - hosts database 2866
 - opening 2866
 - Internet domain names 465, 466
 - compressing 465
 - expanding 466
 - messages 2534, 2537, 2540, 2830, 2834, 2838
 - receiving 2534, 2537, 2540
- TCP/IP (*continued*)
 - messages (*continued*)
 - sending 2830, 2834, 2838
 - network database 522, 2884
 - closing 522
 - opening 2884
 - protocols database 523, 2892
 - closing 523
 - opening 2892
 - services database 525, 2842, 2905
 - closing 525
 - entry structure 2842
 - opening 2905
 - sockets 211
 - ports, binding to 211
- TCSADRAIN 3364
- TCSAFLUSH 3364
- TCSANOW 3364
- TCSBRK 1199
- tcsendbreak() 3362
- TCSETA 1199
- TCSETAF 1199
- tcsetattr() 3364
- TCSETAW 1199
- tcsetpgrp() 3367
- TCSETS 1199
- TCSETSF 1199
- tcsetsid() 3369
- tcsetsize() 3371
- TCSETSW 1199
- TCXONC 1199
- TDP 142
 - asynchronous I/O and 142
- Technical support 44
- tell() 1557
- tell(), tell64() 3373
- telldir() 3375
- tempnam() 3377
- temporary files 1645, 1647, 3377, 3478, 3480
 - creating 3478
 - creating and opening 1645
 - name, generating 1647, 3377, 3480
- TERM 300
- terminal control 2043, 3338, 3340, 3342, 3345, 3349, 3351, 3353, 3355, 3362, 3367, 3371
 - characters, injecting 3355
 - communications line 3340, 3362
 - break condition, asserting 3362
 - disconnecting 3340
 - draining 3338
 - flow control 3342
 - flushing 3345
 - process group ID 2043, 3349, 3351, 3367
 - getting 3349
 - setting 2043, 3367
 - size 3353, 3371
- terminal group, starting new 3095
- terminals 262, 264, 270, 274, 276, 387, 700, 1198, 1427, 1873, 1937, 2509, 3347, 3364, 3369, 3379
 - attributes, setting 270
 - canonical input buffer 700, 1937

terminals (*continued*)

- control 3347, 3364, 3379
 - attributes 3347, 3364
 - structure 3379
- controlling 387, 3369
 - making 3369
 - path name 387
- display, controlling 1873
- file descriptor, testing for association with 1427
- input speed 262, 274
- operating attributes 1198
- output speed 264, 276
- raw input buffer 700, 1937
- reading 2509
- TERMINFO 300
- termios 270, 2510, 3365, 3379
- text segment 237, 549
 - beginning of 237
 - end of 549
- third-party licenses 41
- thread_pool_attr_t 3388
- THREAD_POOL_CONTROL_HIWATER 3385
- THREAD_POOL_CONTROL_INCREMENT 3385
- THREAD_POOL_CONTROL_LOWATER 3385
- THREAD_POOL_CONTROL_MAXIMUM 3385
- THREAD_POOL_CONTROL_NONBLOCK 3386
- thread_pool_control() 3386
- thread_pool_create() 3387
- thread_pool_destroy() 3393
- thread_pool_limits() 3396
- thread_pool_start() 3397
- ThreadCancel(), ThreadCancel_r() 3399
- ThreadCreate(), ThreadCreate_r() 3403
- ThreadCtl() 1172, 2144
- ThreadCtl(), ThreadCtl_r() 247, 254, 1148, 1155, 1160, 1162, 1164, 1166, 1174, 1176, 1179, 1181, 2979, 3408, 3518, 3519
- ThreadDestroy(), ThreadDestroy_r() 3417
- ThreadDetach(), ThreadDetach_r() 3420
- ThreadJoin(), ThreadJoin_r() 3422
- threads 279, 281, 313, 316, 333, 539, 669, 908, 922, 981, 1855, 1857, 1859, 1861, 1863, 1865, 1947, 2041, 2043, 2076, 2105, 2182, 2185, 2187, 2189, 2191, 2193, 2195, 2197, 2199, 2201, 2203, 2205, 2207, 2209, 2211, 2213, 2215, 2217, 2219, 2221, 2223, 2225, 2227, 2229, 2231, 2233, 2235, 2237, 2239, 2241, 2243, 2245, 2247, 2249, 2251, 2253, 2255, 2257, 2259, 2261, 2263, 2265, 2269, 2272, 2274, 2276, 2278, 2281, 2282, 2285, 2288, 2290, 2292, 2294, 2295, 2297, 2299, 2301, 2303, 2305, 2308, 2310, 2312, 2314, 2316, 2318, 2322, 2324, 2327, 2329, 2331, 2333, 2335, 2337, 2339, 2341, 2343, 2345, 2347, 2349, 2352, 2353, 2355, 2358, 2359, 2361, 2364, 2366, 2369, 2371, 2373, 2375, 2377, 2379, 2381, 2383, 2385, 2387, 2389, 2391, 2392, 2394, 2396, 2398, 2400, 2402, 2405, 2407, 2409, 2411, 2413, 2415, 2418, 2419, 2422, 2424, 2426, 2428, 2430, 2432, 2433, 2743, 2745, 2756, 2762, 2763, 2764, 2766, 2770, 2774, 2781, 2783, 2890, 2900, 2951,

threads (*continued*)

- 2962, 2965, 2970, 2983, 2987, 2992, 3004, 3010, 3014, 3028, 3030, 3032, 3034, 3036, 3038, 3040, 3042, 3094, 3095, 3280, 3282, 3286, 3289, 3305, 3309, 3310, 3386, 3387, 3393, 3396, 3397, 3399, 3403, 3404, 3405, 3408, 3409, 3410, 3412, 3413, 3415, 3417, 3420, 3422, 3652, 3659, 3664, 3667
- aborting 2185
- adaptive partitioning 2745
- attributes 908, 2189, 2191, 2193, 2197, 2199, 2201, 2203, 2205, 2207, 2209, 2211, 2213, 2215, 2217, 2221, 2223, 2225, 2227, 2229, 2231, 2233, 2235, 3309, 3403
- contention scope 2201, 2225
- destroying 2189
- detach state 2191, 2215
- guard area, size of 908, 2193, 2217, 3309
- initializing 2213
- scheduling parameters 2197, 2221
- scheduling policy 2199, 2223
- stack address 2203, 2205, 2227, 2229
- stack size 2211, 2235
- stack, lazy 2207, 2209, 2231, 2233
- bankruptcy, notifying of 2766
- barriers 2237, 2239, 2241, 2243, 2245, 2247, 2249
 - attributes 2239, 2243, 2247
 - attributes, process-shared 2245, 2249
 - destroying 2237
 - initializing 2239
 - waiting at 2241
- blocking 3282
- busy-waiting 1857, 1861, 1863, 1865
- calibrating 1859
- cancelling 2251, 3399
- cancellation 2253, 2255, 2392, 2394, 2432, 3404
 - cleanup handlers 2253, 2255
 - points 3404
 - points, creating 2432
 - state 2392
 - type 2394
- clock ID 2295
- concurrency 2294, 2396
- condition variables 2257, 2259, 2261, 2263, 2265, 2269, 2272, 2274, 2276, 2278, 2281, 2282, 3289, 3305
 - attributes 2272, 2278
 - attributes, clock 2274, 2281
 - attributes, process-shared 2276, 2282
 - blocking on 2265, 2269
 - destroying 2259, 3289
 - initializing 2261, 3305
 - unblocking 2257, 2263
- continue, causing to 3412, 3415
- creating 2285, 3403
- critical 2762, 2764, 2965
 - determining if 2764
 - marking as 2762, 2965
- data 2301, 2305, 2308, 2405, 3405
- destroying 3417
- detached 2215, 3403
- detaching from a process 2288, 3420

threads (*continued*)

- errno 539
- events 2766
 - attaching 2766
- execution time 313, 333
- files, locking 669
- fork handlers, registering 2187
- holding 3412, 3415
- I/O privileges, requesting 3409, 3410
- ID, getting 981
- IDs 2182, 2290, 2391
 - calling thread 2182, 2391
 - comparing 2290
- initializing 2361
- joinable 2215, 3403
- joining 2303, 2433, 3422
 - with a time limit 2433
- keys 2305, 2308
- local storage 2301, 2305, 2308, 2405, 3405
- maximum number of 2900
- misaligned access response 3409
- mutexes 2312, 2314, 2316, 2318, 2322, 2324, 2327, 2329, 2331, 2333, 2335, 2337, 2339, 2341, 2343, 2345, 2347, 2349, 2352, 2353, 2355, 2358, 2359, 3286, 3289, 3305
 - attributes, destroying 2333
 - attributes, initializing 2347
 - attributes, priority ceiling 2335, 2349
 - attributes, process-shared 2339, 2353
 - attributes, recursive 2341, 2355
 - attributes, scheduling protocol 2337, 2352
 - attributes, type 2343, 2358
 - attributes, wake-up 2345
 - attributes, wakeup 2359
 - destroying 2312, 3289
 - initializing 2316, 3305
 - locking 2318, 2324, 2327
 - priority ceiling 2314, 2322
 - unlocking 2329
 - waking up blocked threads 2331, 3286
- names 2297, 2398, 3410
- noncritical, marking as 2763, 2965
- once-initialization 2361
- partitions 2756, 2764, 2770
 - determining 2764
 - joining 2756, 2770
- pool 3386, 3387, 3393, 3396, 3397
 - attributes, changing 3386, 3396
 - creating 3387
 - destroying 3393
 - starting 3397
- priority 279, 281, 922, 2318, 2324, 2329, 2402, 2890, 2962
 - getting 922
 - inheritance 279, 2318, 2324, 2329
 - inheritance, preventing 281
 - on receiving a pulse 2962
 - setting 2402, 2890
- private data 2301, 2305, 2308, 2405, 3405
- processor affinity 2041, 2043, 2076, 2105, 3094, 3095, 3412, 3413
- QNX-specific changes, making 3408

threads (*continued*)

- read-write locks 2364, 2366, 2369, 2371, 2373, 2375, 2377, 2379, 2381, 2383, 2385, 2387, 2389, 3289, 3305
 - attributes, creating 2387
 - attributes, destroying 2383
 - attributes, process-shared 2385, 2389
 - destroying 2364, 3289
 - initializing 2366, 3305
 - locking for reading 2369, 2371, 2375
 - locking for writing 2373, 2377, 2381
 - unlocking 2379
- return status 3404
- scheduling parameters 2299, 2400, 2774, 2781
- scheduling policy 2195, 2219, 2299, 2400, 2774, 2781, 3403, 3405
 - inheriting 2195, 2219, 3403
- scope 3403
- signal mask 2970, 2992, 3004, 3010, 3014
 - getting 3004
 - restoring 2970
 - saving 3010
 - setting 3004, 3014
 - signal-blocked 2992
- signals 1947, 2310, 2407, 2951, 2983, 2987, 3404, 3405
 - initial state 3405
 - mask 2407
 - sending 2310
 - targeting 2951, 2983, 2987
 - terminating on 3404
 - waiting for 1947
- sleepon locks 2409, 2411, 2413, 2415, 2418, 2419, 3030, 3032, 3034, 3036, 3038, 3040, 3042
 - destroying 3032
 - initializing 3034
 - locking 2411, 3036
 - unblocking 2409, 2413, 3030, 3038
 - unlocking 2418, 3040
 - waiting 2415, 2419, 3042
- spinlocks 2422, 2424, 2426, 2428, 2430
 - destroying 2422
 - initializing 2424
 - locking 2426, 2428
 - unlocking 2430
- stack 3310, 3404
 - minimum size (`_SC_THREAD_STACK_MIN`) 3310
- suspending 316, 1855, 3014, 3028, 3652, 3659, 3664, 3667
- synchronizing 2241
- system overload, notifying of 2766
- terminating 2292
- terminating unconditionally 2185
- waking up 2409, 2413, 3030, 3038, 3280
- yielding 2743, 2783
- zombies 3403

tick size 329, 330, 2747

- changing 330, 2747
- getting and setting 329

time members 1225

- in attribute structure of resource managers 1225

Time Stamp Counter (TSC) 325

time to live (TTL) 1391
 time zone 346, 3310, 3475, 3546, 3547
 abbreviations 3310, 3546
 maximum length 3310
 default 346
 offset from UTC 3475
 setting 3547
 time_t 1005, 1007, 3425
 tm, converting to/from 1005, 1007
 time() 3425
 timeb 791
 timeout, setting on a blocking state 3443
 TIMER_ABSTIME 315, 3439, 3442, 3460, 3464
 timer_create() 2148, 3428
 timer_delete() 3431
 timer_getexpstatus() 2148, 3433
 timer_getoverrun() 2148, 3435
 timer_gettime() 2148, 3437
 TIMER_PRECISE 3439, 3443, 3460, 3465
 timer_settime() 3439
 timer_timeout(), timer_timeout_r() 3443
 TIMER_TOLERANCE 3439, 3443, 3460, 3465
 TimerAlarm(), TimerAlarm_r() 3449
 TimerCreate() 2148
 TimerCreate(), TimerCreate_r() 3452
 TimerDestroy(), TimerDestroy_r() 3454
 TimerInfo() 2148
 TimerInfo(), TimerInfo_r() 3457
 timers 316, 889, 2872, 3309, 3428, 3431, 3433, 3435,
 3437, 3439, 3449, 3452, 3454, 3457, 3461,
 3465
 alarm, scheduling 3449
 creating 3452
 destroying 3454
 information about, getting 3457
 interval 889, 2872
 value, getting 889
 value, setting 2872
 maximum number of overruns (_SC_DELAYTIMER_MAX)
 3309
 realtime 3428, 3431, 3433, 3435, 3437, 3439, 3461
 creating 3428
 destroying 3431
 expiry status 3433
 overruns 3435
 time until expiry 3437, 3439, 3461
 threads 316
 timeout, setting on kernel blocking state 3465
 TimerSettime(), TimerSettime_r() 3461
 TimerTimeout(), TimerTimeout_r() 3465
 times 155, 305, 307, 310, 313, 319, 323, 325, 327, 329,
 333, 389, 396, 417, 791, 802, 982, 1005, 1007,
 1364, 1375, 1378, 1531, 1533, 1649, 1895,
 2736, 2881, 2915, 3202, 3233, 3327, 3425,
 3473, 3474, 3476, 3547, 3588, 3591, 3684
 booting, since 3327
 calendar 1531, 1533, 1649, 3425, 3476
 current 3425
 local, converting to/from 1531, 1533, 1649
 structure 3476
 clock 307, 310, 313, 319, 323, 325, 327, 329, 333
 adjusting 323
 times (*continued*)
 clock (*continued*)
 cycles 325
 getting 313
 getting and setting 333
 ID, getting 307, 327, 333
 period, getting and setting 329
 resolution, getting 310
 setting 319
 current 791, 982, 2915, 3425
 calendar 3425
 getting 791, 982
 setting 2915
 daylight saving time 396, 3547
 difference, calculating 417
 files 802, 1364, 1375, 1378, 3588, 3591
 access 1364, 1375, 1378, 3591
 modification 802, 1364, 1375, 1378, 3588, 3591
 status-change 1364, 1375, 1378
 formatting 2881, 3202, 3233, 3684
 local 1531, 1533, 1649
 calendar, converting to/from 1531, 1533, 1649
 nanoseconds 1895, 3474
 timespec, converting to/from 1895, 3474
 processes 305, 2736
 execution time limit, getting 2736
 execution time, in clock ticks 305
 specification structure 3473
 time_t 389, 1005, 1007
 strings, converting to/from 389
 tm, converting to/from 1005, 1007
 timespec 1895, 3474
 nanoseconds, converting to/from 1895, 3474
 tm 155
 strings, converting to/from 155
 times() 3471
 timespec 309, 1895, 3473, 3474
 nanoseconds, converting to/from 1895, 3474
 timespec2nsec() 3474
 timezone 3475, 3547
 TIOCCBRK 1199
 TIOCCDTR 1199
 TIOC Drain 1199
 TIOCEXCL 1199
 TIOCF LUSH 1199
 TIOCGETA 1199
 TIOCGETC 1199
 TIOCGETP 1199
 TIOCGETPGRP 1199
 TIOCG LTC 1199
 TIOCGPGRP 1199
 TIOCGSIZE 1199
 TIOCGWINSZ 1199
 TIOCHPCL 1199
 TIOCLGET 1199
 TIOCLSET 1199
 TIOCMBIC 1199
 TIOCMBIS 1199
 TIOCMGET 1199
 TIOCMSET 1199
 TIOCN OTTY 1199
 TIOCNXCL 1199

- TIOCOUTQ 1199
 - TIOCPKT 1199
 - TIOCSBRK 1199
 - TIOCSCTTY 1199
 - TIOCSDTR 1199
 - TIOCSETA 1199
 - TIOCSETAF 1199
 - TIOCSETAW 1199
 - TIOCSETC 1199
 - TIOCSETN 1199
 - TIOCSETP 1199
 - TIOCSETPGRP 1199
 - TIOCSINUSE 1199
 - TIOCSLTC 1199
 - TIOCSPGRP 1199
 - TIOCSSIZE 1199
 - TIOCSTART 1199
 - TIOCSTI 1199
 - TIOCSTOP 1199
 - TIOCSWINSZ 1199
 - tm 155, 389, 1005, 1007, 1531, 1649, 3476, 3684
 - strings, converting to/from 155, 389
 - time_t, converting to/from 1005, 1007
 - TMP_MAX 3378
 - tmpfile(), tmpfile64() 3478
 - tmpnam() 3480
 - tms 3471
 - tokens, breaking a string into 3258, 3260, 3707
 - tolower() 3482
 - TOS (type of service) 1391
 - TOSTOP 3382
 - toupper() 3484
 - towctrans() 3486
 - tolower() 3488
 - toupper() 3490
 - trace events 3492, 3494, 3496, 3498, 3500, 3502, 3504,
3506, 3508, 3510, 3517, 3524, 3526, 3529,
3531, 3533, 3534, 3537
 - callbacks, attaching 3526, 3529
 - class and event type 3517
 - controlling the tracing of 3510
 - CPU index 3517
 - debug mode, setting 3531
 - inserting 3492, 3494, 3496, 3498, 3500, 3502, 3504,
3506, 3508
 - arbitrary class and type 3500
 - current address 3496
 - function entry 3492
 - function exit 3494
 - user combine event 3498
 - user simple event 3504
 - user string event 3502, 3506, 3508
 - parser state structure 3533, 3534, 3537
 - destroying 3533
 - information about 3534
 - initializing 3537
 - parsing 3524
 - trace_func_enter() 3492, 3520
 - trace_func_exit() 3494, 3520
 - trace_here() 3496, 3520
 - trace_logb() 3498, 3521
 - trace_logbc() 3500, 3520
 - trace_logf() 3502, 3521
 - trace_logi() 3504, 3521
 - trace_nlogf() 3506, 3521
 - trace_vnlogf() 3508, 3521
 - TraceEvent() 2149, 3510
 - tracep_callb_func_t 3527
 - traceparser_cs_range() 3529
 - traceparser_cs() 3526
 - traceparser_debug() 3531
 - traceparser_destroy() 3533
 - traceparser_get_info() 3534
 - traceparser_init() 3537
 - traceparser() 3524
 - Transmission Control Protocol, See TCP
 - Transparent Distributed Processing 142
 - asynchronous I/O and 142
 - triggers 2180, 2182
 - getting values of 2180
 - trigonometry 118, 157, 163, 165, 377, 3024, 3334
 - See also hyperbolic functions
 - arccosine 118
 - arcsine 157
 - arctangent 163, 165
 - cosine 377
 - sine 3024
 - tangent 3334
 - See also hyperbolic functions
 - truncate(), truncate64() 3539
 - TRY_AGAIN 1013, 1018
 - TSC (Time Stamp Counter) 325
 - TTL (time to live) 1391
 - ttynam_r() 3544
 - ttynam() 3542
 - type of service (TOS) 1391
 - typed memory 1656
 - mapping 1656
 - typed memory objects 2122, 2124
 - opening 2124
 - querying 2122
 - Typographical conventions 42
 - TZ 300, 3547
 - tzname 3546, 3547
 - TZNAME_MAX 3310
 - tzset() 1650, 3203, 3547
- ## U
- ualarm() 3550
 - UDP (User Datagram Protocol) 3553, 3765
 - not supported by SOCKS 3765
 - UIO_MAXIOV 2523, 2525, 3746
 - UIOCCMD 1199
 - ulltoa() 3555
 - ultoa() 3555
 - umask() 2149, 3558
 - umount() 3559
 - UNALIGNED_PUT16() 3561
 - UNALIGNED_PUT32() 3563
 - UNALIGNED_PUT64() 3565
 - UNALIGNED_RET16() 3567
 - UNALIGNED_RET32() 3568
 - UNALIGNED_RET64() 3569

uname() 3571
 ungetc() 3573
 ungetwc() 3575
 union, offset of members within 1900
 Unix classification 49
 UNIX-domain protocol 913, 3577
 peer credentials 913
 unlink() 3579
 unlockpt() 3582
 UNMAP_INIT_OPTIONAL 1839
 UNMAP_INIT_REQUIRED 1839
 unsetenv() 3584
 uppercase 1453, 1477, 3266, 3484, 3486, 3490, 3720
 characters, converting to 3484
 strings, converting to 3266
 testing a character for 1453, 1477
 wide characters, converting to 3486, 3490, 3720
 usage of system resources 945
 User Datagram Protocol (UDP) 3553, 3765
 not supported by SOCKS 3765
 user information file 527, 985, 987, 989, 2462, 2919, 3594, 3596
 closing 527
 entry 3594
 reading 985, 989
 renaming 3596
 returning to beginning of 2919
 searching 987
 writing 2462
 USER_PROCESS 987, 3595
 users 840, 891, 892, 934, 937, 938, 941, 984, 2039, 2041, 2042, 2043, 2066, 2095, 2856, 2897, 2900, 2917, 3308, 3309
 IDs 840, 984, 2039, 2041, 2042, 2043, 2066, 2095, 2856, 2897, 2917, 3309
 effective 840, 2039, 2041, 2042, 2043, 2856, 2897, 2917
 real 984, 2039, 2041, 2042, 2043, 2066, 2095, 2897, 2917
 saved 2917
 set-user 2041, 3309
 names 891, 892
 password database, getting entry for 934, 937, 938, 941
 processes, maximum per real user ID 2900, 3308
 usleep() 3586
 utilities, locating 345
 utimbuf 802, 3588
 utime() 1375, 1378, 3588
 resource managers, implementing in 1375, 1378
 utimes() 3591
 utmp 3594
 utmpname() 3596
 utoa() 3598
 utsname 3571

V

va_arg() 3602
 va_copy() 3605
 va_end() 3607
 va_start() 3608
 valloc() 3610

variable-length argument lists ("varargs") 3602, 3605, 3607, 3608
 variables, global 151, 153, 154, 199, 237, 396, 491, 494, 539, 540, 549, 903, 904, 2183, 2978, 3175, 3176, 3177, 3329, 3475, 3546
 __progname 2183
 _ambblksiz 151
 _argc 153
 _argv 154
 _auxv 199
 _btext 237
 _edata 491
 _end 494
 _etext 549
 _syspage_ptr 3329
 daylight 396
 errno 539
 optarg 904
 opterr 904
 optind 903
 optopt 904
 stderr 3175
 stdin 3176
 stdout 3177
 sys_errlist 539
 sys_nerr 540
 sys_nsig 2978
 sys_siglist 2978
 timezone 3475
 tzname 3546
 verr(), verrx() 3612
 vfork() 3614
 vfprintf() 3616
 vfscanf() 3618
 vfwprintf() 3620
 vfwscanf() 3622
 virtual 8086 mode 1185
 void pointers, size of 51
 vprintf() 3624
 vscanf() 3626
 vslog2f() 3629
 vslogf() 3631
 vsnprintf() 3633
 vsprintf() 3635
 vsscanf() 3637
 vswprintf() 3639
 vswscanf() 3641
 vsyslog() 3643
 vwarn(), vwarnx() 3645
 vwprintf() 3647
 wscanf() 3649

W

W_OK 61, 488
 wait() 2149, 3472, 3652, 3659, 3667
 wait3() 2149, 3656
 wait4() 2149, 3659
 waitfor() 3661
 waitid() 2149, 3664
 waitpid() 2149, 3472, 3667
 warn(), warnx() 3669

- warnings, formatted on stderr 3645, 3669
- WCONTINUED 3655, 3658, 3663, 3666
- WCOREDUMP() 3653
- wcrtomb() 3671
- wcscat() 3673
- wcschr() 3674
- wcscmp() 3676
- wcscoll() 3678, 3715
- wcscpy() 3680
- wcscspn() 3682
- wcsftime() 3684
- wcslen() 3686
- wcsncat() 3687
- wcsncmp() 3689
- wcsncpy() 3691
- wcspbrk() 3693
- wcsrchr() 3695
- wcsrtombs() 3697
- wcsspn() 3699
- wcssstr() 3701
- wctod() 3703
- wctof() 3703
- wctoimax() 3706
- wctok() 3707
- wctol() 3709
- wctold() 3703
- wctoll() 3709
- wctombs() 3711
- wctoul(), wctoull() 3713
- wctoumax() 3706
- wctob() 3717
- wctomb() 3718
- wctrans() 3720
- wctype() 3722
- WEXITED 3655, 3658, 3663, 3666
- WEXITSTATUS() 3331, 3653
- whitespace, testing a character for 1451, 1475
- wide characters 41, 238, 656, 658, 711, 713, 804, 806, 810, 993, 995, 1455, 1457, 1459, 1461, 1463, 1465, 1467, 1469, 1471, 1473, 1475, 1477, 1479, 1587, 1589, 1591, 1593, 1595, 2466, 2468, 3272, 3274, 3486, 3488, 3490, 3575, 3620, 3622, 3639, 3647, 3649, 3671, 3673, 3674, 3676, 3678, 3680, 3682, 3686, 3687, 3689, 3691, 3693, 3695, 3697, 3699, 3701, 3703, 3706, 3707, 3709, 3711, 3713, 3715, 3717, 3718, 3720, 3722, 3724, 3726, 3728, 3730, 3732, 3736, 3748
 - See also characters; strings
 - classes 3722
 - converting 3486, 3720
 - copying 3728, 3730
 - overlapping objects 3730
 - lowercase, converting to 3486, 3488, 3720
 - multibyte characters, converting to/from 1587, 1589, 1595, 3671, 3718
 - conversion object, status of 1589
 - searching 3674, 3695, 3724
 - in a string 3674, 3695
 - in memory 3724
 - sets of, searching for 3682, 3693, 3699
 - setting 3732
- wide characters (*continued*)
 - single-byte characters, converting to/from 238, 3717
 - stdin, reading from 995
 - stdout, writing to 2468
 - streams 656, 658, 711, 804, 993, 2466, 3575
 - orientation 804
 - pushing back 3575
 - reading 656, 658, 993
 - writing to 711, 2466
 - strings 713, 806, 810, 1591, 1593, 3272, 3274, 3620, 3622, 3639, 3647, 3649, 3673, 3674, 3676, 3678, 3680, 3682, 3686, 3687, 3689, 3691, 3693, 3695, 3697, 3699, 3701, 3703, 3706, 3707, 3709, 3711, 3713, 3715, 3726, 3736, 3748
 - comparing 3676, 3678, 3689, 3726
 - concatenating 3673, 3687
 - copying 3680, 3691, 3715
 - formatted 3272, 3639
 - input, formatted 810, 3274, 3622, 3649, 3748
 - length 3686
 - multibyte characters, converting to/from 1591, 1593, 3697, 3711
 - numbers, converting to/from 3703, 3706, 3709, 3713
 - output, formatted 806, 3620, 3647, 3736
 - searching for a set of wide characters 3682, 3693, 3699
 - searching for a string 3701
 - searching for a wide character 3674, 3695
 - splitting into tokens 3707
 - streams, writing to 713
 - transforming 3715
 - testing for 1455, 1457, 1459, 1461, 1463, 1465, 1467, 1469, 1471, 1473, 1475, 1477, 1479
 - alphabetic 1457
 - alphanumeric 1455
 - blank 1459
 - character class 1463
 - control character 1461
 - decimal digit 1465
 - hexadecimal digit 1479
 - lowercase 1469
 - printable 1467, 1471
 - punctuation 1473
 - uppercase 1477
 - whitespace 1475
 - uppercase, converting to 3486, 3490, 3720
 - See also characters; strings
- wide mode, controlling 3514
- WIFCONTINUED() 3653
- WIFEXITED() 3653
- WIFSIGNALED() 3653
- WIFSTOPPED() 3653
- wmemchr() 3724
- wmemcmp() 3726
- wmemcpy() 3728
- wmemmove() 3730
- wmemset() 3732
- WNOHANG 3655, 3658, 3663, 3666
- WNOWAIT 3655, 3658, 3663, 3666
- word expansions 3734, 3735

wordexp() 3734
wordfree() 3735
working directory 286, 610, 830, 997
wprintf() 3736
WRDE_NOSYS 3734
Write Combining 2927
write() 1225, 1380, 3737
 resource managers, implementing in 1380
writeblock() 3742
writev() 3745
wscanf() 3748
WSTOPPED 3655, 3659, 3663, 3667
WSTOPSIG() 3653
WTERMSIG() 3653
WUNTRACED 3655, 3659, 3664, 3667

X

X_OK 61, 488

Y

y0(), y0f() 3754
y1(), y1f() 3756
yn(), ynf() 3758

Z

zombies 2041, 2043, 2950, 2982, 3094, 3403
 preventing children from becoming 2041, 2043, 2950,
 2982, 3094
 threads 3403

