Core Networking Stack User's Guide

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/

Electronic edition published: Friday, March 14, 2014

Table of Contents

About This Guide

This guide introduces you to the QNX Neutrino Core Networking stack and its manager, `io-pkt`.

The following table may help you find information quickly in this guide:

For information on:	Go to:
`io-pkt` and its architecture	*Overview* (p. 9)
Examining and modifying packets, and creating a firewall	*Packet Filtering* (p. 21)
Setting up secure connections	*IP Security and Hardware Encryption* (p. 29)
802.11 a/b/g (Wi-Fi)	*Wi-Fi Configuration Using WPA and WEP* (p. 37)
`io-pkt` and Qnet	*Transparent Distributed Processing* (p. 63)
Support for different types of network drivers	*Network Drivers* (p. 65)
Related utilities, etc.	*Utilities, Managers, and Configuration Files*
How to write your own network driver	*Writing Network Drivers for `io-pkt`*
Sample code for the above	*A Hardware-Independent Sample Driver: `sam.c`*
More information about writing a network driver	*Additional Information*
Terms used in this document	*Glossary*

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	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl** –**Alt** –**Delete**
Keyboard input	`Username`
Keyboard keys	`Enter`
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
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
Overview

The QNX Neutrino networking stack is called `io-pkt`. It replaces the previous generation of the stack, `io-net`, and provides the following benefits:

- performance improvements. Since `io-pkt` removes the `npkt`-to-`mbuf` translation and mandatory queuing, and also reduces context switching on the packet receive path, the IP receive performance is greatly improved.
- simplified locking of shared resources, resulting in simpler SMP support
- it closely follows the NetBSD code base and architecture, meaning:
    - easier maintenance / upgrade capability of IP stack source
    - existing applications that use BSD standard APIs will port more easily (e.g. `tcpdump`)
    - enhanced features included with NetBSD stack are also included with `io-pkt`
    - NetBSD drivers will port in a straightforward manner
- far richer stack feature set, drawing on the latest in improvements from the NetBSD code base
- 802.11 Wi-Fi client and access point capability

The `io-pkt` manager is intended to be a drop-in replacement for `io-net` for those people who are dealing with the stack from an outside application point of view. It includes stack variants, associated utilities, protocols, libraries and drivers.

The stack variants are:

`io-pkt-v4`

IPv4 version of the stack with no encryption or Wi-Fi capability built in. This is a “reduced footprint” version of the stack that doesn't support the following:

- IPv6
- Crypto / IPSec
- 802.11 a/b/g Wi-Fi
- Bridging
- GRE / GRF
- Multicast routing
- Multipoint PPP

`io-pkt-v4-hc`

IPv4 version of the stack that has full encryption and Wi-Fi capability built in and includes hardware-accelerated cryptography capability (Fast IPsec).

io-pkt-v6-hc

IPv6 version of the stack (includes IPv4 as part of v6) that has full encryption and Wi-Fi capability, also with hardware-accelerated cryptography.

In this guide, we use "`io-pkt`" to refer to *all* the stack variants. When you start the stack, use the appropriate variant (`io-pkt` isn't a symbolic link to any of them).

We've designed `io-pkt` to follow as closely as possible the NetBSD networking stack code base and architecture. This provides an optimal path between the IP protocol and drivers, tightly integrating the IP layer with the rest of the stack.

The `io-pkt` implementation makes significant changes to the QNX Neutrino stack architecture, including the following:

- `io-net` is replaced by the stack's link layer
- `mbufs` are used throughout, including in the drivers
- all buffer management is handled by the stack
- `mount` and `umount` capabilities:
  - Only `io-net` drivers may be both mounted and unmounted. Other drivers may allow you to detach the driver from the stack, by using the `ifconfig` *iface* `destroy` command (if the driver supports it).
  - The IP stack is an integral part of `io-pkt`; you can't start `io-pkt` without it. This means that you don't need to specify the -ptcpip option to the stack unless there are additional parameters (e.g. prefix=) that you need to pass to it. If you specify the -ptcpip option without additional parameters, `io-pkt` accepts it with no effect.
  - Protocols and enhanced stack functionality (e.g. TDP, NAT / IP Filtering) can be mounted, but not unmounted.
- The concepts of producers and consumers no longer exist within `io-pkt`. Filters still exist, but they use a different API than with `io-net`. The `io-pkt` stack does provide other hooks into the stack that you can use to provide a similar level of functionality. These include:

  Berkeley Packet Filter interface

  Lets you read and write, but not modify or discard, both IP and Ethernet packets from your application.

  Packet Filter interface

`pfil` hooks, enabled when the PF filter module is loaded, let you read, write, and modify IP and Ethernet packets within the context of the stack process.

- Driver changes:
  - The driver model has changed to provide better integration with the protocol stack. (For example, in `io-net`, `npkts` had to be converted into `mbufs` for use with the stack. In `io-pkt`, `mbufs` are used throughout.)
  - The driver API and behavior have been changed to closely match those of the NetBSD stack, allowing NetBSD drivers to be ported to `io-pkt`.
  - A shim layer, `devnp-shim.so`, is provided that lets you use an `io-net` driver as-is.
  - By default, driver interfaces are no longer sequentially numbered with `en`*x* designations; they're named according to driver type (e.g. `fxp0` is the interface for an Intel 10/100 driver). You can use the name= driver option (processed by `io-pkt`) to specify the interface name.
  - Drivers no longer present entries in the name space to be directly opened and accessed with a *devctl()* command (e.g. `open(/dev/io-net/en0)`). Instead, a socket file descriptor is opened and queried for interface information. The *ioctl()* command is then sent to the stack using this device information.
- IP Filtering and NAT are now handled through the PF interface with `pfctl`. This replaces the `io-net` `ipf` interface, which is no longer supported.
- SCTP isn't supported in the initial release of `io-pkt`.
- In `io-net`, loopback-checksumming was done with `ifconfig`. In `io-pkt` this is controlled via `sysctl`:

```
# sysctl -a | grep do_loopback_cksum
net.inet.ip.do_loopback_cksum = 0
net.inet.tcp.do_loopback_cksum = 0
net.inet.udp.do_loopback_cksum = 0
```

- The `nicinfo` utility operates slightly differently from the way it did under `io-net`:
  - The default operation with no arguments is to list information on all interfaces. The behavior with `io-net` was to show stats for `/dev/io-net/en0`.
  - Under `io-net`, all Ethernet interface names were in the form `en`*X*. Under `io-pkt`, this name will vary, but you can use the name= driver option (processed by `io-pkt`) to override this.
  - Ported NetBSD drivers might not support the `nicinfo` *ioctl()* command.

Architecture of `io-pkt`

The `io-pkt` stack is very similar in architecture to other component subsystems inside of the QNX Neutrino operating system.

At the bottom layer are drivers that provide the mechanism for passing data to, and receiving data from, the hardware. The drivers hook into a multi-threaded layer-2 component (that also provides fast forwarding and bridging capability) that ties them together and provides a unified interface into the layer-3 component, which then handles the individual IP and upper-layer protocol-processing components (TCP and UDP).

In the QNX Neutrino RTOS, a resource manager forms a layer on top of the stack. The resource manager acts as the message-passing intermediary between the stack and user applications. It provides a standardized type of interface involving *open()*, *read()*, *write()*, and *ioctl()* that uses a message stream to communicate with networking applications. Networking applications written by the user link with the socket library. The socket library converts the message-passing interface exposed by the stack into a standard BSD-style socket layer API, which is the standard for most networking code today.

One of the big differences that you'll see with this stack as compared to `io-net` is that it isn't currently possible to decouple the layer 2 component from the IP stack. This was a trade-off that we made to allow increased performance at the expense of some reduced versatility. We might look at enabling this at some point in the future if there's enough demand.

In addition to the socket-level API, there are also other, programmatic interfaces into the stack that are provided for other protocols or filtering to occur. These interfaces—used directly by Transparent Distributed Processing (TDP, also known as Qnet)—are very different from those provided by `io-net`, so anyone using similar interfaces to these in the past will have to rewrite them for `io-pkt`.

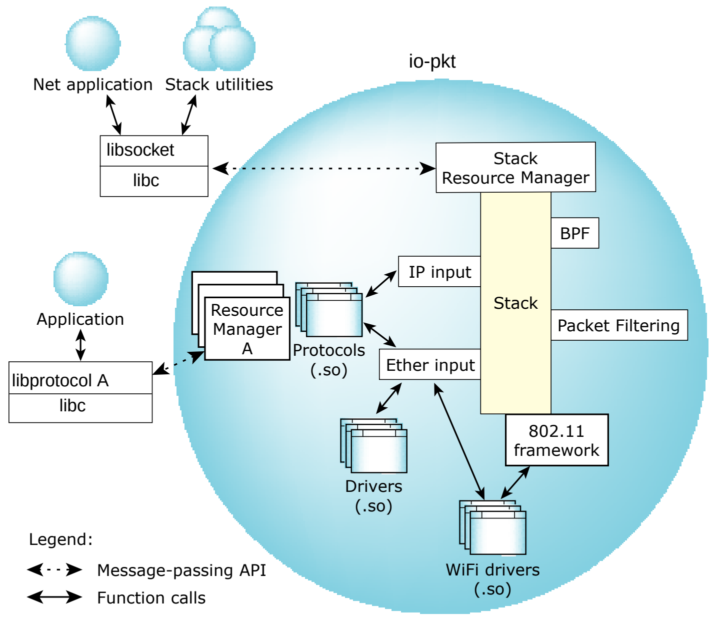

Figure 1: A detailed view of the `io-pkt` architecture.

At the driver layer, there are interfaces for Ethernet traffic (used by all Ethernet drivers), and an interface into the stack for 802.11 management frames from wireless drivers. The `hc` variants of the stack also include a separate hardware crypto API that allows the stack to use a crypto offload engine when it's encrypting or decrypting data for secure links. You can load drivers (built as DLLs for dynamic linking and prefixed with `devnp-`) into the stack using the -d option to `io-pkt`.

APIs providing connection into the data flow at either the Ethernet or IP layer allow protocols to coexist within the stack process. Protocols (such as Qnet) are also built as DLLs. A protocol links directly into either the IP or Ethernet layer and runs within the stack context. They're prefixed with `lsm` (loadable shared module) and you load them into the stack using the -p option. The `tcpip` protocol (-ptcpip) is a special option that the stack recognizes, but doesn't link a protocol module for (since the IP stack is already present). You still use the -ptcpip option to pass additional parameters to the stack that apply to the IP protocol layer (e.g., -ptcpip prefix=/alt to get the IP stack to register `/alt/dev/socket` as the name of its resource manager).

A protocol requiring interaction from an application sitting outside of the stack process may include its own resource manager infrastructure (this is what Qnet does) to allow communication and configuration to occur.

In addition to drivers and protocols, the stack also includes hooks for packet filtering. The main interfaces supported for filtering are:

Berkeley Packet Filter (BPF) interface

A socket-level interface that lets you read and write, but not modify or block, packets, and that you access by using a socket interface at the application layer (see *http://en.wikipedia.org/wiki/Berkeley_Packet_Filter*). This is the interface of choice for basic, raw packet interception and transmission and gives applications outside of the stack process domain access to raw data streams.

Packet Filter (PF) interface

A read/write/modify/block interface that gives complete control over which packets are received by or transmitted from the upper layers and is more closely related to the `io-net` filter API.

For more information, see the *Packet Filtering* (p. 21) chapter.

Threading model

The default mode of operation is for `io-pkt` to create one thread per CPU. The `io-pkt` stack is fully multi-threaded at layer 2.

However, only one thread may acquire the "stack context" for upper-layer packet processing. If multiple interrupt sources require servicing at the same time, these may be serviced by multiple threads. Only one thread will service a particular interrupt source at any point in time. Typically an interrupt on a network device indicates that there are packets to be received. The same thread that handles the receive processing may later transmit the received packets out another interface. Examples of this are layer-2 bridging and the "ipflow" fastforwarding of IP packets.

The stack uses a thread pool to service events that are generated from other parts of the system. These events include:

- time-outs
- ISR events
- other things generated by the stack or protocol modules

You can use a command-line option to the driver to control the priority of threads that receive packets. Client connection requests are handled in a floating priority mode (i.e. the thread priority matches that of the client application thread accessing the stack resource manager).

Once a thread receives an event, it examines the event type to see if it's a hardware event, stack event, or "other" event:

- If the event is a hardware event, the hardware is serviced and, for a receive packet, the thread determines whether bridging or fast-forwarding is required. If so, the thread performs the appropriate lookup to determine which interface the packet should be queued for, and then takes care of transmitting it, after which it goes back to check and see if the hardware needs to be serviced again.
- If the packet is meant for the local stack, the thread queues the packet on the stack queue. The thread then goes back and continues checking and servicing hardware events until there are no more events.
- Once a thread has completed servicing the hardware, it checks to see if there's currently a stack thread running to service stack events that may have been generated as a result of its actions. If there's no stack thread running, the thread *becomes* the stack thread and loops, processing stack events until there are none remaining. It then returns to the "wait for event" state in the thread pool.

This capability of having a thread change directly from being a hardware-servicing thread to being the stack thread eliminates context switching and greatly improves the receive performance for locally terminated IP flows.

If `io-pkt` runs out of threads, it sends a message to `slogger`, and anything that requires a thread blocks until one becomes available. You can use command-line options to specify the maximum and minimum number of threads for `io-pkt`.

Threading priorities

There are a couple of ways that you can change the priority of the threads responsible for receiving packets from the hardware.

You can pass the rx_prio_pulse option to the stack to set the default thread priority. For example:

```
io-pkt-v4 -ptcpip rx_pulse_prio=50
```

This makes all the receive threads run at priority 50. The current default for these threads is priority 21.

The second mechanism lets you change the priority on a *per-interface* basis. This is an option passed to the driver and, as such, is supported only if the driver supports it. When the driver registers for its receive interrupt, it can specify a priority for the pulse that is returned from the ISR. This pulse priority is what the thread will use when running. Here's some sample code for *my_board*:

```
if ((rc = interrupt_entry_init(&my_board->inter_rx, 0, NULL,
    cfg->priority)) != EOK) {
        log(LOG_ERR, "%s(): interrupt_entry_init(rx) failed: %d",
           __FUNCTION__, rc);
        my_board_destroy(my_board, 9);
                return rc;
}
```

Driver-specific thread priorities are assigned on a *per-interface* basis. The stack normally creates one thread per CPU to allow the stack to scale appropriately in terms of performance on an SMP system. Once you use an interface-specific parameter with multiple interfaces, you must get the stack to create one thread per interface in order to have that option picked up and used properly by the stack. This is handled with the -t option to the stack.

For example, to have the stack start up and receive packets on one interface at priority 20 and on a second interface at priority 50 on a single-processor system, you would use the following command-line options:

```
io-pkt-v4 -t2 -dmy_board syspage=1,priority=20,pci=0 \
-dmy_board syspage=1,priority=50,pci=1
```

If you've specified a per-interface priority, and there are more interfaces than threads, the stack sends a warning to `slogger`. If there are insufficient threads present, the per-interface priority is ignored (but the rx_pulse_prio option is still honored).

The actual options for setting the priority and selecting an individual card depend on the device driver; see the driver documentation for specific option information.

Legacy `io-net` drivers create their own receive thread, and therefore don't require the -t option to be used if they support the priority option. These drivers use the `devnp-shim.so` shim driver to allow interoperability with the `io-pkt` stack.

Components of core networking

The `io-pkt` manager is the main component.

Other core components include:

`pfctl`, `lsm-pf-v6.so`, `lsm-pf-v4.so`

IP Filtering and NAT configuration and support.

`ifconfig`, `netstat`, *`sockstat`* (see the NetBSD documentation), `sysctl`

Stack configuration and parameter / information display.

`pfctl`

Priority packet queuing on Tx (QoS).

`lsm-autoip.so`

Auto-IP interface configuration protocol.

`lsm-slip.so`, `slattach`

Serial Line IP (SLIP) network interface.

`brconfig`

Bridging and STP configuration along with other layer-2 capabilities.

`pppd`, `pppoectl`

PPP support for `io-pkt`, including PPP, PPPOE (client), and Multilink PPP.

`devnp-shim.so`

`io-net` binary-compatibility shim layer.

`nicinfo`

Driver information display tool (for native and `io-net` drivers).

`libsocket.so`

BSD socket application API into the network stack.

`libpcap.so`, `tcpdump`

Low-level packet-capture capability that provides an abstraction layer into the Berkeley Packet Filter interface.

`lsm-qnet.so`

Transparent Distributed Processing protocol for `io-pkt`.

hostapd, *hostapd_cli* (see the NetBSD documentation), wpa_supplicant, wpa_cli

Authentication daemons and configuration utilities for wireless access points and clients.

QNX Neutrino Core Networking also includes applications, services, and libraries that interface to the stack through the socket library and are therefore not directly dependent on the Core components. This means that they use the standard BSD socket interfaces (BSD socket API, Routing Socket, PF_KEY, raw socket):

libssl.so, libssl.a

SSL suite ported from the source at *http://www.openssl.org*.

libnbdrvr.so

BSD porting library. An abstraction layer provided to allow the porting of NetBSD drivers.

libipsec(S).a, setkey

NetBSD IPsec tools.

inetd

Updated Internet daemon.

route

Updated route-configuration utility.

ping, ping6

Updated `ping` utilities.

ftp, ftpd

Enhanced FTP.

Chapter 2
Packet Filtering

In principle, the pseudo-devices involved with packet filtering are as follows:

- `pf` is involved in filtering network traffic
- `bpf` is an interface that captures and accesses raw network traffic.

The `pf` pseudo-device is implemented using `pfil` hooks; `bpf` is implemented as a tap in all the network drivers. We'll discuss them briefly from the point of view of their attachment to the rest of the stack.

If you're using QNX Neutrino 6.4.1 or earlier, you should use *ioctl_socket()* instead of *ioctl()* in your packet-filtering code. With the microkernel message-passing architecture, *ioctl()* calls that have pointers embedded in them need to be handled specially. The *ioctl_socket()* function uses *ioctl()* for functionality that doesn't require special handling.

In QNX Neutrino 6.5.0 and later, *ioctl()* handles embedded pointers, so you don't have to use *ioctl_socket()* instead.

Packet Filter interface

The pfil interface is purely in the stack and supports packet-filtering hooks.

Packet filters can register hooks that are called when packet processing is taking place; in essence, pfil is a list of callbacks for certain events. In addition to being able to register a filter for incoming and outgoing packets, pfil provides support for interface attach/detach and address change notifications.

The pfil interface is one of a number of different layers that a user-supplied application can register for, to operate within the stack process context. These modules, when compiled, are called Loadable Shared Modules (lsm) in QNX Neutrino nomenclature, or Loadable Kernel Modules (lkm) in BSD nomenclature.

There are two levels of registration required with io-pkt:

- The first allows the user-supplied module to connect into the io-pkt framework and access the stack infrastructure.
- The second is the standard NetBSD mechanism for registering functions with the appropriate layer that sends and receives packets.

In the QNX Neutrino RTOS, shared modules are dynamically loaded into the stack. You can do this by specifying them on the command line when you start io-pkt, using the -p option, or you can add them subsequently to an existing io-pkt process by using the mount command.

The application module must include an initial module entry point defined as follows:

```
#include "sys/io-pkt.h"
#include "nw_datastruct.h"

int mod_entry( void *dll_hdl, struct _iopkt_self *iopkt,
               char *options)
{

}
```

The calling parameters to the entry function are:

void * *dll_hdl*

An opaque pointer that identifies the shared module within io-pkt.

struct _iopkt_self * *iopkt*

A structure used by the stack to reference its own internals.

char * *options*

The options string passed by the user to be parsed by this module.

The header files aren't installed as part of the OS. If you need them, contact your sales representative.

This is followed by the registration structure that the stack will look for after calling *dlopen()* to load the module to retrieve the entry point:

```
struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(mod) =
  IOPKT_LSM_ENTRY_SYM_INIT(mod_entry);
```

This entry point registration is used by all shared modules, regardless of which layer the remainder of the code is going to hook into. Use the following functions to register with the `pfil` layer:

```
#include <sys/param.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>

struct pfil_head *
pfil_head_get(int af, u_long dlt);

struct packet_filter_hook *
pfil_hook_get(int dir, struct pfil_head *ph);

int
pfil_add_hook(int (*func)(), void *arg, int flags,
              struct pfil_head *ph);

int
pfil_remove_hook(int (*func)(), void *arg, int flags,
                 struct pfil_head *ph);

int
(*func)(void *arg, struct mbuf **mp, struct ifnet *, int dir);
```

The *head_get()* function returns the start of the appropriate `pfil` hook list used for the hook functions. The *af* argument can be either `PFIL_TYPE_AF` (for an address family hook) or `PFIL_TYPE_IFNET` (for an interface hook) for the "standard" interfaces.

If you specify `PFIL_TYPE_AF`, the Data Link Type (*dlt*) argument is a protocol family. The current implementation has filtering points for only `AF_INET` (IPv4) or `AF_INET6` (IPv6).

When you use the interface hook (`PFIL_TYPE_IFNET`), *dlt* is a pointer to a network interface structure. All hooks attached in this case will be in reference to the specified network interface.

Once you've selected the appropriate list head, you can use *pfil_add_hook()* to add a hook to the filter list. This function takes as arguments a filter hook function, an opaque pointer (which is passed into the user-supplied filter *arg* function), a *flags* value (described below), and the associated list head returned by *pfil_head_get()*.

The *flags* value indicates when the hook function should be called and may be one of:

- `PFIL_IN` — call me on incoming packets.
- `PFIL_OUT` — call me on outgoing packets.

- PFIL_ALL — call me on all of the above.

When a filter is invoked, the packet appears just as if it came off the wire. That is, all protocol fields are in network-byte order. The filter returns a nonzero value if the packet processing is to stop, or zero if the processing is to continue.

For interface hooks, the *flags* argument can be one of:

- PFIL_IFADDR — call me when the interface is reconfigured (mbuf ** is an *ioctl()* number).
- PFIL_IFNET — call me when the interface is attached or detached (mbuf ** is either PFIL_IFNET_ATTACH or PFIL_IFNET_DETACH)

Here's an example of what a simple pfil hook would look like. It shows when an interface is attached or detached. Upon a detach (ifconfig *iface* destroy), the filter is unloaded.

```
#include <sys/types.h>
#include <errno.h>
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/socket.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include "sys/io-pkt.h"
#include "nw_datastruct.h"


static int in_bytes = 0;
static int out_bytes = 0;


static int input_hook(void *arg, struct mbuf **m,
                      struct ifnet *ifp, int dir)
{
    in_bytes += (*m)->m_len;
    return 0;
}

static int output_hook(void *arg, struct mbuf **m,
                       struct ifnet *ifp, int dir)
{
    out_bytes += (*m)->m_len;
    return 0;
}


static int deinit_module(void);
static int iface_hook(void *arg, struct mbuf **m,
                      struct ifnet *ifp, int dir)
{
    printf("Iface hook called ... ");
    if ( (int)m == PFIL_IFNET_ATTACH) {
        printf("Interface attached\n");
        printf("%d bytes in, %d bytes out\n", in_bytes,
               out_bytes);
    } else if ((int)m == PFIL_IFNET_DETACH) {
        printf("Interface detached\n");
        printf("%d bytes in, %d bytes out\n", in_bytes,
               out_bytes);
        deinit_module();
    }
    return 0;
}

static int ifacecfg_hook(void *arg, struct mbuf **m,
                         struct ifnet *ifp, int dir)
```

```
{
    printf("Iface cfg hook called with 0x%08X\n", (int)(m));

    return 0;
}

static int deinit_module(void)
{
    struct pfil_head *pfh_inet;

    pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
    if (pfh_inet == NULL) {
        return ESRCH;
    }
    pfil_remove_hook(input_hook, NULL, PFIL_IN | PFIL_WAITOK,
                     pfh_inet);
    pfil_remove_hook(output_hook, NULL, PFIL_OUT | PFIL_WAITOK,
                     pfh_inet);

    pfh_inet = pfil_head_get(PFIL_TYPE_IFNET, 0);
    if (pfh_inet == NULL) {
        return ESRCH;
    }

    pfil_remove_hook(ifacecfg_hook, NULL, PFIL_IFNET, pfh_inet);

    pfil_remove_hook(iface_hook, NULL, PFIL_IFNET | PFIL_WAITOK,
                     pfh_inet);
    printf("Unloaded pfil hook\n" );

    return 0;
}


int pfil_entry(void *dll_hdl, struct _iopkt_self *iopkt,
               char *options)
{
    struct pfil_head *pfh_inet;

    pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
    if (pfh_inet == NULL) {
        return ESRCH;
    }
    pfil_add_hook(input_hook, NULL, PFIL_IN | PFIL_WAITOK,
                  pfh_inet);
    pfil_add_hook(output_hook, NULL, PFIL_OUT | PFIL_WAITOK,
                  pfh_inet);

    pfh_inet = pfil_head_get(PFIL_TYPE_IFNET,0);
    if (pfh_inet == NULL) {
        return ESRCH;
    }

    pfil_add_hook(iface_hook, NULL, PFIL_IFNET, pfh_inet);
    pfil_add_hook(ifacecfg_hook, NULL, PFIL_IFADDR, pfh_inet);
    printf("Loaded pfil hook\n" );

    return 0;
}

struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(pfil) =
  IOPKT_LSM_ENTRY_SYM_INIT(pfil_entry);
```

Packet Filter (`pf`) module: firewalls and NAT

The `pfil` interface is used by the Packet Filter (`pf`) to hook into the packet stream for implementing firewalls and NAT. This is a loadable module specific to either the v4 or v6 version of the stack (`lsm-pf-v4.so` or `lsm-pf-v6.so`). When loaded (e.g.

`mount -Tio-pkt /lib/dll/lsm-pf-v4.so`), the module creates a `pf` pseudo-device.

The `pf` pseudo-device provides roughly the same functionality as `ipfilter`, another filtering and NAT suite that also uses the `pfil` hooks.

For more information, see the following in the *Utilities Reference*:

`pf`

Packet Filter pseudo-device

`pf.conf`

Configuration file for `pf`

`pfctl`

Control the packet filter and network address translation (NAT) device

To start `pf`, use the `pfctl` utility, which issues a `DIOCSTART` *ioctl()* command. This causes `pf` to call *pf_pfil_attach()*, which runs the necessary `pfil` attachment routines. The key routines after this are *pf_test()* and *pf_test6()*, which are called for IPv4 and IPv6 packets respectively. These functions test which packets should be sent, received, or dropped. The packet filter hooks, and therefore the whole of `pf`, are disabled with the `DIOCSTOP` *ioctl()* command, usually issued with `pfctl -d`.

For more information about using PF, see `pf-faq` at *ftp://ftp3.usa.openbsd.org/pub/OpenBSD/doc/* in the OpenBSD documentation. Certain portions of the document (related to packet queueing, CARP and others) don't apply to our stack, but the general configuration information is relevant. This document covers both firewalling and NAT configurations that you can apply using PF.

Berkeley Packet Filter

The Berkeley Packet Filter (BPF) provides link-layer access to data available on the network through interfaces attached to the system.

To use BPF, open a device node, `/dev/bpf`, and then issue *ioctl()* commands to control the operation of the device. A popular example of a tool using BPF is `tcpdump` (see the *Utilities Reference*).

The device `/dev/bpf` is a cloning device, meaning you can open it multiple times. It is in principle similar to a cloning interface, except BPF provides no network interface, only a method to open the same device multiple times.

To capture network traffic, you must attach a BPF device to an interface. The traffic on this interface is then passed to BPF for evaluation. To attach an interface to an open BPF device, use the `BIOCSETIF` *ioctl()* command. The interface is identified by passing a `struct ifreq`, which contains the interface name in ASCII encoding. This is used to find the interface from the kernel tables. BPF registers itself to the interface's `struct ifnet` field, *if_bpf*, to inform the system that it's interested in traffic on this particular interface. The listener can also pass a set of filtering rules to capture only certain packets, for example ones matching a given combination of host and port.

BPF captures packets by supplying a *bpf_tap()* tapping interface to link layer drivers, and by relying on the drivers to always pass packets to it. Drivers honor this request and commonly have code which, along both the input and output paths, does:

```
#if NBPFILTER > 0
    if (ifp->if_bpf)
        bpf_mtap(ifp->if_bpf, m0);
#endif
```

This passes the `mbuf` to the BPF for inspection. BPF inspects the data and decides if anyone listening to this particular interface is interested in it. The filter inspecting the data is highly optimized to minimize the time spent inspecting each packet. If the filter matches, the packet is copied to await being read from the device.

The BPF tapping feature and the interfaces provided by `pfil` provide similar services, but their functionality is disjoint. The BPF mtap wants to access packets right off the wire without any alteration and possibly copy them for further use. Callers linking into `pfil` want to modify and possibly drop packets. The `pfil` interface is more analogous to `io-net`'s filter interface.

BPF has quite a rich and complex syntax (e.g. *http://www.rawether.net/support/bpfhelp.htm*) and is a standard interface that is used by a lot of networking software. It should be your interface of first choice when packet interception / transmission is required. It will also be a slightly lower performance interface given that it does operate across process boundaries with filtered packets being copied before being passed outside of the stack domain and into the application domain. The `tcpdump` and `libpcap` library operate using the BPF interface to intercept and display packet activity. For those of you currently using something like the `nfm-nraw` interface in `io-net`, BPF provides the equivalent functionality, with some extra complexity involved in setting things up, but with much more versatility in configuration.

Chapter 3
IP Security and Hardware Encryption

The io-pkt-v4-hc and io-pkt-v6-hc stack variants include full, built-in support for IPsec.

> You need to specify the ipsec parameter option to the stack in order to enable IPsec when the stack starts.

There's a good reference page in the NetBSD man pages covering IPsec in general. There are some aspects that don't apply (you obviously don't have to worry about rebuilding the kernel), but the general usage is the same.

Setting up an IPsec connection: examples

The following examples illustrate how to set up IPsec:

- *between two boxes manually* (p. 30)
- *with authentication using the preshared-key method* (p. 31)

Between two boxes manually

Suppose we have two boxes, A and B, and we want to establish IPsec between them.

Here's how:

1. On each box, create a script file (let's say its name is `my_script`) having the following content:

```
#!/bin/ksh
# args: This script takes two arguments:
#    - The first one is the IP address of the box that is to
#      run it on.
#    - The second one is the IP address of the box that this
#      box is to establish IPsec connection to.
Myself=$1
Remote=$2

# The following two lines are to clean the database.
# They're here simply to demonstrate the "hello world" level
# connection.
#
setkey -FP
setkey -F

# Use setkey to input all of the SA content.
setkey -c << EOF

spdadd $Myself $Remote any -P out ipsec
esp/transport/$Myself-$Remote/require;
spdadd $Remote $Myself any -P in ipsec esp/transport/$Remote-$Myself/require;

add  $Myself $Remote esp 1234 -m any -E 3des-cbc "KeyIsTwentyFourBytesLong";
add  $Remote $Myself esp 1234 -m any -E 3des-cbc "KeyIsTwentyFourBytesLong";
EOF
```

2. On BoxA, run `./my_script BoxA BoxB`, or give the IP address of each box if the name can't be resolved.
3. Similarly, on BoxB, run `./my_script BoxB BoxA`.

Now you can check the connection by pinging each box from the other. You can get the IPsec status by using `setkey -PD`.

With authentication using the preshared-key method

Consider the simplest case where there are two boxes, BoxA and BoxB. User A is on BoxA, User B is on Box B, and the two users have a shared secret, which is a string of `hello_world`.

1. On Box A, create a file, `psk.txt`, that has these related lines:

   ```
   usera@qnx.com   "Hello_world"
   userb@qnx.com   "Hello_world"
   ```

   The IPsec IKE daemon, `racoon`, will use this file to do the authentication and IPsec connection job.

2. The `root` user must own `psk.txt` and the file's permissions must be read/write only by `root`. To ensure this is the case, run:

   ```
   chmod 0600 psk.txt
   ```

3. The `racoon` daemon needs a configuration file (e.g., `racoon.conf`) that defines the way that `racoon` is to operate. In the remote session, specify that we're going to use the preshared key method as authentication and let `racoon` know where to find the secret. For example:

   ```
   ...
   # Let racoon know where your preshared keys are:

   path pre_shared_key "your_full_path_to_psk.txt" ;
   remote anonymous
   {
       exchange_mode aggressive,main;
       doi ipsec_doi;
       situation identity_only;

       #my_identifier address;
       my_identifier user_fqdn "usera@qnx.com";
       peers_identifier user_fqdn "userb@qnx.com";

       nonce_size 16;
       lifetime time 1 hour;   # sec,min,hour
       initial_contact on;
       proposal_check obey;    # obey, strict or claim

       proposal {
           encryption_algorithm 3des;
           hash_algorithm sha1;
           authentication_method pre_shared_key ;
           dh_group 2 ;
       }
   }
   ...
   ```

4. Set up the policy using `setkey`. You can use the following script (called `my_script`) to tell the stack that the IPsec between BoxA and BoxB requires key negotiation:

   ```
   #!/bin/sh
   # This is a simple configuration for testing racoon negotiation.
   #

   Myself=$1
   Remote=$2
   ```

```
setkey -FP
setkey -F
setkey -c << EOF
#
spdadd $Remote $Myself any -P in  ipsec
esp/transport/$Remote-$Myself/require;
spdadd $Myself $Remote any -P out ipsec
esp/transport/$Myself-$Remote/require;
#
EOF
```

Run this on BoxA as `./my_script BoxA BoxB`.

5. Repeat the above steps on BoxB. Needless to say, on BoxB you need to run as `./my_script BoxB BoxA` (and so on).
6. On both boxes, run `racoon -c full_path_to_racoon.conf`. When you initiate traffic, say by trying to `ping` the peer box, `racoon` will do its job and establish the IPsec connection by creating Security Associations (SAs) for both directions, and then you can see the traffic passing back and forth, which indicates that the IPsec connection is established.

IPsec tools

The QNX Neutrino Core Networking uses the IPsec tools from the NetBSD source base and incorporates it into its source base.

The tools include:

`libipsec`

PF_KEY library routines.

`setkey`

Security Policy Database and Security Association Database management tool.

`racoon`

IKE key-management daemon. This utility is available only in binary form on request. Under encryption export law, we must track to whom we send this technology and report the information to the US government.

`racoonctl`

A command-line tool that controls `racoon`.

OpenSSL support

We've ported the OpenSSL crypto and SSL libraries (from *http://www.openssl.org*) for your applications to use.

Hardware-accelerated crypto

The `io-pkt-v4-hc` and `io-pkt-v6-hc` managers have the (hardware-independent) infrastructure to load a (hardware-dependent) driver to take advantage of dedicated hardware that can perform cryptographic operations at high speed. This not only speeds up the crypto operations (such as those used by IPsec), but also reduces the CPU load.

This interface is carefully crafted so that the stack doesn't block on the crypto operation; rather, it continues, and later on, using a callback, the driver returns the processed data to the stack. This is ideal for DMA-driven crypto hardware.

Chapter 4
Wi-Fi Configuration Using WPA and WEP

802.11 a/b/g Wi-Fi capability is built into the two "hc" variants of the stack (`io-pkt-v4-hc` and `io-pkt-v6-hc`). The NetBSD stack includes its own separate 802.11 MAC layer that's independent of the driver. Many other implementations pull the 802.11 MAC inside the driver; as a result, every driver needs separate interfaces and configuration utilities. If you write a driver that conforms to the stack's 802.11 layer, you can use the same set of configuration and control utilities for all wireless drivers.

The networking Wi-Fi solution lets you join or host WLAN (Wireless LAN) networks based on IEEE 802.11 specifications. Using `io-pkt`, you can:

- connect using a peer-to-peer mode called *ad hoc mode*, also referred to as *Independent Basic Service Set (IBSS)* configuration
- either act as a client for a Wireless Access Point (WAP, also known as a *base station*) or configure QNX Neutrino to act as a WAP. This second mode is referred to as *infrastructure mode* or *BSS (Basic Service Set)*.

Ad hoc mode lets you create a wireless network quickly by allowing wireless nodes within range (for example, the wireless devices in a room) to communicate directly with each other without the need for a wireless access point. While being easy to construct, it may not be appropriate for a large number of nodes because of performance degradation, limited range, non-central administration, and weak encryption.

Infrastructure mode is the more common network configuration where all wireless hosts (clients) connect to the wireless network via a WAP (Wireless Access Point). The WAP centrally controls access and authentication to the wireless network and provides access to rest of your network. More than one WAP can exist on a wireless network to service large numbers of wireless clients.

The `io-pkt` manager supports WEP, WPA, WPA2, or no security for authentication and encryption when acting as the WAP or client. WPA/WPA2 is the recommended encryption protocol for use with your wireless network. WEP isn't as secure as WPA/WPA2 and is known to be breakable. It's available for backward compatibility with already deployed wireless networks.

For information on connecting your client, see "*Using `wpa_supplicant` to manage your wireless network connections* (p. 51)" later in this chapter.

NetBSD 802.11 layer

The `net80211` layer provides functionality required by wireless cards. The code is meant to be shared between FreeBSD and NetBSD, and you should try to keep NetBSD-specific bits in the source file `ieee80211_netbsd.c` (likewise, there's `ieee80211_freebsd.c` in FreeBSD).

For more information about the `ieee80211` interfaces, see Chapter 9 (Kernel Internals) of the NetBSD manual pages at *http://www.netbsd.org/docs/*.

The responsibilities of the `net80211` layer are as follows:

- MAC-address-based access control
- crypto
- input and output frame handling
- node management
- radiotap framework for `bpf` and `tcpdump`
- rate adaption
- supplementary routines, such as conversion functions and resource management

The `ieee80211` layer positions itself logically between the device driver and the ethernet module, although for transmission it's called indirectly by the device driver instead of control passing straight through it. For input, the `ieee80211` layer receives packets from the device driver, strips any information useful only to wireless devices, and in case of data payload proceeds to hand the Ethernet frame up to `ether_input`.

Device management

The way to describe an `ieee80211` device to the `ieee80211` layer is by using a `struct ieee80211com`, declared in `<sys/net80211/ieee80211_var.h>`.

You use it to register a device to the `ieee80211` from the device driver by calling *ieee80211_ifattach()*. Fill in the underlying `struct ifnet` pointer, function callbacks, and device-capability flags. If a device is detached, the `ieee80211` layer can be notified with *ieee80211_ifdetach()*.

Nodes

A node represents another entity in the wireless network. It's usually a base station when operating in BSS mode, but can also represent entities in an ad hoc network.

A node is described by a `struct ieee80211_node`, declared in `<sys/net80211/ieee80211_node.h>`. This structure includes the node unicast encryption key, current transmit power, the negotiated rate set, and various statistics.

A list of all the nodes seen by a certain device is kept in the `struct ieee80211com` instance in the field *ic_sta* and can be manipulated with the helper functions provided in `sys/net80211/ieee80211_node.c`. The functions include, for example, methods to scan for nodes, iterate through the node list, and functionality for maintaining the network structure.

Crypto support

Crypto support enables the encryption and decryption of the network frames.

It provides a framework for multiple encryption methods, such as WEP and null crypto. Crypto keys are mostly managed through the *ioctl()* interface and inside the `ieee80211` layer, and the only time that drivers need to worry about them is in the send routine when they must test for an encapsulation requirement and call *ieee80211_crypto_encap()* if necessary.

Using Wi-Fi with `io-pkt`

When you're connecting to a Wireless Network in the QNX Neutrino RTOS, the first step that you need to do is to start the stack process with the appropriate driver for the installed hardware.

For information on the available drivers, see the `devnp-*` entries in the *Utilities Reference*. For this example, we'll use the fictitious `devnp-abc.so` driver. After a default installation, all driver binaries are installed under the staging directory /*cpu*/`lib/dll`.

The `io-pkt-v4` stack variant doesn't have the 802.11 layer built in, and therefore you can't use it with Wi-Fi drivers. If you attempt to load a Wi-Fi driver into `io-pkt-v4`, you'll see a number of unresolved symbol errors, and the driver won't work.

In this example, start the stack using one of these commands:

- `io-pkt-v4-hc -d /lib/dll/devnp-abc.so`

  or:

- `io-pkt-v6-hc -d abc`

If the network driver is installed in a location other than `/lib/dll`, you'll need to specify the full path and filename of the driver on the command line.

Once you've started the stack and appropriate driver, you need to determine what wireless networks are available. If you already have the name (SSID or Service Set Identifier) of the network you want to join, you can skip these steps. You can also use these steps to determine if the network you wish to join is within range and active:

1. To determine which wireless networks are available to join, you must first set the interface status to up:

   ```
   ifconfig abc0 up
   ```

2. Check to see which wireless networks have advertised themselves:

   ```
   wlanctl abc0
   ```

   This command lists the available networks and their configurations. You can use this information to determine the network name (SSID), its mode of operation (ad hoc or infrastructure mode), and radio channel, for example.

3. You can also force a manual scan of the network with this command:

   ```
   ifconfig abc0 scan
   ```

This will cause the wireless adapter to scan for WAP stations or ad hoc nodes within range of the wireless adapter, and list the available networks, along with their configurations. You can also get scan information from the `wpa_supplicant` utility (described later in this document).

Once you've started the appropriate driver and located the wireless network, you'll need to choose the network mode to use (ad hoc or infrastructure mode), the authentication method to attach to the wireless network, and the encryption protocol (if any) to use.

We recommend that you implement encryption on your wireless network if you aren't using any physical security solutions.

By default, most network drivers will infrastructure mode (BSS), because most wireless networks are configured to allow network access via a WAP. If you wish to implement an ad hoc network, you can change the network mode by using the `ifconfig` command:

- To create or join ad hoc networks, use:

  ```
  ifconfig abc0 mediaopt adhoc
  ```

- If you wish to switch back to infrastructure mode, you can use this command to connect to WAP on Infrastructure networks (note the minus sign in front of the `mediaopt` command):

  ```
  ifconfig abc0 -mediaopt adhoc
  ```

For information about your driver's media options, see its entry in the *Utilities Reference*. When you're in ad hoc mode, you advertise your presence to other peers that are within physical range. This means that other 802.11 devices can discover you and connect to your network.

Whether you're a client in infrastructure mode, or you're using ad hoc mode, the steps to implement encryption are the same. You need to make sure that you're using the authentication method and encryption key that have been chosen for the network. If you wish to connect with your peers using an ad hoc wireless network, all peers must be using the same authentication method and encryption key. If you're a client connecting to a WAP, you must use the same authentication method and encryption key as have been configured on the WAP.

Connecting to a wireless network

For the general case of connecting to a Wi-Fi network, we recommend that you use the `wpa_supplicant` daemon.

It handles unsecure, WEP, WPA, and WPA2 networks and provides a mechanism for saving network information in a configuration file that's scanned on startup, thereby removing the need for you to constantly reenter network parameters after rebooting or moving from one network domain to another. The information following covers the more specific cases if you don't want to run the supplicant.

You can connect to a wireless network by using one of the following:

- *no encryption* (p. 42)
- *WEP (Wired Equivalent Privacy) for authentication and encryption* (p. 43)
- *WPA/WPA2 for authentication and encryption* (p. 45)
- *`wpa_supplicant` to manage your wireless network connections* (p. 51)

Once connected, you need to configure the interface in the standard way:

- TCP/IP configuration in a wireless network (client in Infrastructure Mode, or ad hoc Mode)

Using no encryption

If you're creating a wireless network with no encryption, anyone who's within range of the wireless network (e.g. someone driving by your building) can easily view all network traffic. It's possible to create a network without using encryption, but we don't recommend it unless the network has been secured by some other mechanism.

Many consumer devices (wireless routers to connect your internal LAN to the Internet for example) are shipped with security features such as encryption turned off. We recommend that you enable encryption in these devices rather than turn off encryption when creating a wireless network.

To connect using no encryption or authentication, type:

```
ifconfig abc0 ssid "network name" -nwkey
```

The -nwkey option disables WEP encryption and also deletes the temporary WEP key.

The `io-pkt` manager doesn't support a combination of Shared Key Authentication (SKA) and WEP encryption disabled.

Once you've entered the network name, the 802.11 network should be active. You can verify this with `ifconfig`. In the case of ad hoc networks, the status will be shown as active only if there's at least one other peer on the (SSID) network:

```
ifconfig abc0

abc0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500%%
    ssid "network name" %%
    powersave off %%
    bssid 00:11:22:33:44:55 chan 11%%
    address: 11:44:88:44:88:44%%
    media: IEEE802.11 autoselect (OFDM36 mode 11g) %%
    status: active%%
```

Once the network status is active, you can send and receive packets on the wireless link.

You can also use `wpa_supplicant` to associate with a security-disabled Wi-Fi network. For example, if your `/etc/wpa_supplicant.conf` file can contain a network block as follows:

```
network = {
    ssid = "network name"
    key_mgmt = NONE
}
```

you can then run:

```
wpa_supplicant -i abc0 -c/etc/wpa_supplicant.conf
```

You may also use `wpa_cli` to tell `wpa_supplicant` what you want to do. You can use either `ifconfig` or `wpa_cli` to check the status of the network. To complete your network configuration, see "*Client in infrastructure or ad hoc mode* (p. 58)" in the section on TCP/IP interface configuration.

Using WEP (Wired Equivalent Privacy) for authentication and encryption

WEP can be used for both authentication and privacy with your wireless network. Authentication is a required precursor to allowing a station to associate with an access point.

The IEEE 802.11 standard defines the following types of WEP authentication:

Open system authentication

The client is *always* authenticated with the WAP (i.e. allowed to form an association). Keys that are passed into the client aren't checked to see if they're valid. This can have the peculiar effect of having the client interface go "active" (become associated), *but* data won't be passed between the AP and station if the station key used to encrypt the data doesn't match that of the station.

> If your WEP station is active, but no traffic seems to be going through (e.g., `dhcp.client` doesn't work), check the key used for bringing up the connection.

Shared key authentication

This method involves a challenge-response handshake in which a challenge message is encrypted by the stations keys and returned to the access point for verification. If the encrypted challenge doesn't match that expected by the access point, then the station is prevented from forming an association.

Unfortunately, this mechanism (in which the challenge and subsequent encrypted response are available over the air) exposes information that could leave the system more open to attacks, so we don't recommended you use it. While the stack does support this mode of operation, the code hasn't been added to `ifconfig` to allow it to be set.

Note that many access points offer the capability of entering a passphrase that can be used to generate the associated WEP keys. The key-generation algorithm may vary from vendor to vendor. In these cases, the generated hexadecimal keys *must* be used for the network key (prefaced by `0x` when used with `ifconfig`) and not the passphrase. This is in contrast to access points, which let you enter keys in ASCII. The conversion to the hexadecimal key in that case is a simple conversion of the text into its corresponding ASCII hexadecimal representation. The stack supports this form of conversion.

Given the problems with WEP in general, we recommend you use WPA / WPA2 for authentication and encryption where possible.

The network name can be up to 32 characters long. The WEP key must be either 40 bits long or 104 bits long. This means you have to give either 5 or 13 characters for the WEP key, or a 10- or 26-digit hexadecimal value.

You can use either `ifconfig` or `wpa_supplicant` to configure a WEP network.

If you use `ifconfig`, the command is in the form:

`ifconfig` *if_name* `ssid` *the_ssid* `nwkey` *the_key*

For example, if your interface is `abc0`, and you're using 128-bit WEP encryption, you can run:

```
ifconfig abc0 ssid "corporate lan" nwkey corpseckey456 up
```

Once you've entered the network name and encryption method, the 802.11 network should be active (you can verify this with `ifconfig`). In the case of ad hoc networks, the status will be shown as active only if there's at least one other peer on the (SSID) network:

```
ifconfig abc0
abc0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ssid "corporate lan" nwkey corpseckey456
    powersave off
    bssid 00:11:22:33:44:55 chan 11
    address: 11:44:88:44:88:44
    media: IEEE802.11 autoselect (OFDM36 mode 11g)
    status: active
```

Once the network status is active, you can send and receive packets on the wireless link.

If you use `wpa_supplicant`, you need to edit a configuration file to tell it what you want to do. For example:

```
network = {
  ssid = "corporate lan"      # The Wi-Fi network you want to associate to.
  key_mgmt= NONE              # NONE is for WEP or no security.
  wep_key0 = "corpseckey456"  # Most of the time, you may specify a list
                              # from wep_key0 to wep_key3 and use
                              # key index to specify which one to use.
}
```

Then you may run:

```
wpa_supplicant -i abc0 -c your_config_file
```

By default, the configuration file is `/etc/wpa_supplicant.conf`. Alternatively you may use `wpa_cli` to tell the `wpa_supplicant` daemon what you want to do. To complete your network configuration, see "*Client in Infrastructure or ad hoc mode* (p. 58)" in the section on TCP/IP interface configuration.

Using WPA/WPA2 for authentication and encryption

The original security mechanism of the IEEE 802.11 standard wasn't designed to be strong and has proven to be insufficient for most networks that require some kind of security.

Task group I (Security) of the IEEE 802.11 working group (*http://www.ieee802.org/11/*) has worked to address the flaws of the base standard and has in practice completed its work in May 2004. The IEEE 802.11i amendment to the IEEE 802.11 standard was approved in June 2004 and published in July 2004.

The Wi-Fi Alliance used a draft version of the IEEE 802.11i work (draft 3.0) to define a subset of the security enhancements, called Wi-Fi Protected Access (WPA), that can be implemented with existing WLAN hardware. This has now become a mandatory component of interoperability testing and certification done by Wi-Fi Alliance. Wi-Fi provides information about WPA at its website, *http://www.wi-fi.org/*.

The IEEE 802.11 standard defined a Wired Equivalent Privacy (WEP) algorithm for protecting wireless networks. WEP uses RC4 with 40-bit keys, a 24-bit initialization vector (IV), and CRC32 to protect against packet forgery. All these choices have proven to be insufficient:

- The key space is too small to guard against current attacks.
- RC4 key scheduling is insufficient (the beginning of the pseudo-random stream should be skipped).
- The IV space is too small, and IV reuse makes attacks easier.
- There's no replay protection.
- Non-keyed authentication doesn't protect against bit-flipping packet data.

WPA is an intermediate solution for these security issues. It uses the Temporal Key Integrity Protocol (TKIP) to replace WEP. TKIP is a compromise on strong security, and it's possible to use existing hardware. It still uses RC4 for the encryption as WEP does, but with per-packet RC4 keys. In addition, it implements replay protection and a keyed packet-authentication mechanism.

Keys can be managed using two different mechanisms; WPA can use either of the following:

WPA-Enterprise

An external authentication server (e.g. RADIUS) and EAP, just as IEEE 802.1X is using.

WPA-Personal

Pre-shared keys without the need for additional servers.

Both mechanisms generate a master session key for the Authenticator (AP) and Supplicant (client station).

WPA implements a new key handshake (4-Way Handshake and Group Key Handshake) for generating and exchanging data encryption keys between the Authenticator and Supplicant. This handshake is also used to verify that both Authenticator and Supplicant know the master session key. These handshakes are identical regardless of the selected key management mechanism (only the method for generating master session key changes).

WPA utilities

The `wlconfig` library is a generic configuration library that interfaces to the supplicant and provides a programmatic interface for configuring your wireless connection.

The main utilities required for Wi-Fi usage are:

`wpa_supplicant`

Wi-Fi Protected Access client and IEEE 802.1X supplicant. This daemon provides client-side authentication, key management, and network persistence.

The `wpa_supplicant` requires the following libraries and binaries be present:

- `libcrypto.so` — crypto library
- `libssl.so` — Secure Socket Library (created from OpenSSL)
- `random`—executable that creates `/dev/urandom` for random-number generation
- `libm.so` — math library required by `random`
- `libz.so` — compression library required by `random`

The wpa_supplicant also needs a read/write filesystem for creation of a ctrl_interface directory (see the sample wpa_supplicant.conf configuration file).

> You can't use /dev/shmem because it isn't possible to create a directory there.

wpa_cli

WPA command-line client for interacting with wpa_supplicant.

wpa_passphrase

Set the WPA passphrase for a SSID.

hostapd

Server side (Access Point) authentication and key-management daemon.

There are also some subsidiary utilities that you likely won't need to use:

wiconfig

Configuration utility for some wireless drivers. The ifconfig utility can handle the device configuration required without needing this utility.

wlanctl

Examine the IEEE 802.11 wireless LAN client/peer table.

Connecting with WPA or WPA2

Core Networking supports connecting to a wireless network using the more secure option of WPA (Wi-Fi Protected Access) or WPA2 (802.11i) protocols.

The wpa_supplicant application can manage your connection to a single access point, or it can manage a configuration that includes settings for connections to multiple wireless networks (SSIDs) either implementing WPA, or WEP to support roaming from network to network. The wpa_supplicant application supports IEEE802.1X EAP Authentication (referred to as WPA), WPA-PSK, and WPA-NONE (for ad hoc networks) key-management protocols along with encryption support for TKIP and AES (CCMP). A WAP for a simple home or small office wireless network would likely use WPA-PSK for the key-management protocol, while a large office network would use WAP along with a central authentication server such as RADIUS.

To enable a wireless client (or supplicant) to connect to a WAP configured to use WPA, you must first determine the network name (as described above) and get the authentication and encryption methods used from your network administrator. The `wpa_supplicant` application uses a configuration file (`/etc/wpa_supplicant.conf` by default) to configure its settings, and then runs as a daemon in the background. You can also use the `wpa_cli` utility to change the configuration of `wpa_supplicant` while it's running. Changes done by the `wpa_cli` utility are saved in the `/etc/wpa_supplicant.conf` file.

The `/etc/wpa_supplicant.conf` file has a rich set of options that you can configure, but `wpa_supplicant` also uses various default settings that help simplify your wireless configuration. For more information, see *http://netbsd.gw.com/cgi-bin/man-cgi?wpa_supplicant.conf++NetBSD-4.0*.

If you're connecting to a WAP, and your WPA configuration consists of a network name (SSID) and a pre-shared key, your `/etc/wpa_supplicant.conf` would look like this:

```
network={
    ssid="my_network_name"  #The name of the network you wish to join
    psk="1234567890"        #The preshared key applied by the access point
}
```

Make sure that only `root` can read and write this file, because it contains the key information in clear text.

Start `wpa_supplicant` as:

```
wpa_supplicant -B -i abc0 -c /etc/wpa_supplicant.conf
```

The -i option specifies the network interface, and -B causes the application to run in the background.

The `wpa_supplicant` application by default negotiates the use of the WPA protocol, WPA-PSK for key-management, and TKIP or AES for encryption. It uses infrastructure mode by default.

Once the interface status is active (use `ifconfig abc0`, where `abc0` is the interface name, to check), you can apply the appropriate TCP/IP configuration. For more information, see "*TCP/IP configuration in a wireless network* (p. 58)," later in this chapter.

If you were to create an ad hoc network using WPA, your `/etc/wpa_supplicant.conf` file would look like this:

```
network={
    mode=1                   # This sets the mode to be ad hoc.
                             # 0 represents Infrastructure mode
    ssid="my_network_name"   # The name of the ad hoc network
    key_mgmt=NONE            # Sets WPA-NONE
    group=CCMP               # Use AES encryption
    psk="1234567890"         # The preshared key applied by the access point
 }
```

Again, make sure that this file is readable and writable only by `root`, because it contains the key information in clear text.

Start wpa_supplicant with:

```
wpa_supplicant -B -i abc0 -c /etc/wpa_supplicant.conf
```

where -i specifies the network interface, and -B causes the application to run in the background.

Personal-level authentication and Enterprise-level authentication

WPA is designed to have the following authentication methods:

- WPA-Personal / WPA2-Personal, which uses a preshared key that's the same passphrase shared by all network users
- WPA-Enterprise / WPA2-Enterprise, which uses an 802.1X authentication RADIUS-based server to authenticate each user

This section is about the Enterprise-level authentication.

The Enterprise-level authentication methods that have been selected for use within the Wi-Fi certification body are:

- EAP-TLS, which is the initially certified method. Both the server's certificates and the user's certificates are needed.
- EAP-TTLS/MSCHAPv2: TTLS is short for “Tunnelled TLS.” It works by first authenticating the server to the user via its CA certificate. The server and the user then establish a secure connection (the tunnel), and through the secure tunnel, the user gets authenticated. There are many ways of authenticating the user through the tunnel. The EAP-TTLS/MSCHAPv2 uses MSCHAPv2 for this authentication.
- PEAP/MSCHAPv2: PEAP is the secondmost widely supported EAP after EAP-TLS. It's similar to EAP-TTLS, however, it requires only a server-side CA certificate to create a secure tunnel to protect the user authentication. Again, there are many ways of authenticating the user through the tunnel. The PEAP/MSCHAPV2 again uses MSCHAPV2 for authentication.
- PEAP/GTC: This uses GTC as the authentication method through the PEAP tunnel.
- EAP-SIM: This is for the GSM mobile telecom industry.

The io-pkt manager supports all the above, except for EAP-SIM. Certificates are placed in /etc/cert/user.pem, and CA certificates in /etc/cert/root.pem. The following example is the network definition for wpa_supplicant for each of the above Enterprise-level authentication methods:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
update_config=1

# 3.1.2 linksys -- WEP
network={
    ssid="linksys"
    key_mgmt=NONE
    wep_key0="LINKSYSWEPKEY"
}
```

```
# 3.1.3 linksys -- WPA
network={
    ssid="linksys"
    key_mgmt=WPA-PSK
    psk="LINKSYSWPAKEY"
}

# 3.1.4 linksys -- WPA2
network={
    ssid="linksys"
    proto=RSN
    key_mgmt=WPA-PSK
    psk="LINKSYS_RSN_KEY"
}

# 3.1.5.1 linksys -- EAP-TLS
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=TLS
   identity="client1"
   ca_cert="/etc/cert/root.pem"
   client_cert="/etc/cert/client1.pem"
   private_key="/etc/cert/client1.pem"
   private_key_passwd="wzhang"
}


# 3.1.5.2 linksys -- PEAPv1/EAP-GTC
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=PEAP
   identity="client1"
   password="wzhang"
   ca_cert="/etc/cert/root.pem"
   phase1="peaplabel=0"
   phase2="autheap=GTC"
}

# 3.1.5.3 linksys -- EAP-TTLS/MSCHAPv2
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=TTLS
   identity="client1"
   password="wzhang"
   ca_cert="/etc/cert/root.pem"
   phase2="autheap=MSCHAPV2"
}

# 3.1.5.4 linksys -- PEAPv1/EAP-MSCHAPV2
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=PEAP
   identity="client1"
   password="wzhang"
   ca_cert="/etc/cert/root.pem"
   phase1="peaplabel=0"
   phase2="auth=MSCHAPV2"
}
```

Run `wpa_supplicant` as follows:

```
wpa_supplicant -i if_name -c full_path_to_your_config_file
```

to pick up the configuration file and get the supplicant to perform the required authentication to get access to the Wi-Fi network.

Using wpa_supplicant to manage your wireless network connections

The wpa_supplicant daemon is the “standard” mechanism used to provide persistence of wireless networking information as well as manage automated connections into networks without user intervention.

The supplicant is based on the open-source supplicant (albeit an earlier revision that matches that used by the NetBSD distribution) located at *http://hostap.epitest.fi/wpa_supplicant/*.

In order to support wireless connectivity, the supplicant:

- provides a consistent interface for configuring all authentication and encryption mechanisms (unsecure, wep, WPA, WPA2)
- supports configuration of ad hoc and infrastructure modes of operation
- maintains the network configuration information in a configuration file (by default /etc/wpa_supplicant.conf)
- provides auto-connectivity capability allowing a client to connect into a WAP without user intervention

A sample wpa_supplicant.conf file is installed in /etc for you. It contains a detailed description of the basic supplicant configuration parameters and network parameter descriptions (and there are lots of them) and sample network configuration blocks.

In conjunction with the supplicant is a command-line configuration tool called wpa_cli. This tool lets you query the stack for information on wireless networks, as well as update the configuration file on the fly.

If you want wpa_cli to be capable of updating the wpa_supplicant.conf file, edit the file and uncomment the update_config=1 option. (Note that when wpa_cli rewrites the configuration file, it strips all of the comments.) Copy the file into /etc and make sure that root owns it and is the only user who can read or write it, because it contains clear-text keys and password information.

Given a system with a USB-Wi-Fi dongle based on the fictitious ABC chips, here's a sample session showing how to get things working with a WEP-based WAP:

```
# cp $HOME/stage/etc/wpa_supplicant.conf /etc
# chown root:root /etc/wpa_supplicant.conf
# chmod 600 /etc/wpa_supplicant.conf
# io-pkt-v4-hc -dabc
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
    inet 127.0.0.1 netmask 0xff000000
abc0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
    ssid ""
    powersave off
    address: 00:ab:cd:ef:d7:ac
    media: IEEE802.11 autoselect
    status: no network
# wpa_supplicant -B -iabc0
# wpa_cli
wpa_cli v0.4.9
Copyright (c) 2004-2005, Jouni Malinen <jkmaline@cc.hut.fi> and contributors
```

```
This program is free software. You can distribute it and/or modify it
under the terms of the GNU General Public License version 2.

Alternatively, this software may be distributed under the terms of the
BSD license. See README and COPYING for more details.


Selected interface 'abc0'

Interactive mode

> scan
OK
> scan_results
bssid / frequency / signal level / flags / ssid
00:02:34:45:65:76   2437    10  [WPA-EAP-CCMP]  A_NET
00:23:44:44:55:66   2412    10  [WPA-PSK-CCMP]  AN_OTHERNET
00:12:4c:56:a7:8c   2412    10  [WEP]   MY_NET
> list_networks
network id / ssid / bssid / flags
0   simple  any
1   second ssid any
2   example any
> remove_network 0
OK
> remove_network 1
OK
> remove_network 2
OK
> add_network
0
> set_network 0 ssid "MY_NET"
OK
> set_network 0 key_mgmt NONE
OK
> set_network 0 wep_key0 "My_Net_Key234"
OK
> enable_network 0
OK
> save
OK
> list_network
network id / ssid / bssid / flags
0   QWA_NET any
> status
<2>Trying to associate with 00:12:4c:56:a7:8c (SSID='MY_NET' freq=2412 MHz)
<2>Trying to associate with 00:12:4c:56:a7:8c (SSID='MY_NET' freq=2412 MHz)
wpa_state=ASSOCIATING
> status
<2>Trying to associate with 00:12:4c:56:a7:8c (SSID='MY_NET' freq=2462 MHz)
<2>Associated with 00:12:4c:56:a7:8c
<2>CTRL-EVENT-CONNECTED - Connection to 00:12:4c:56:a7:8c completed (auth)
bssid=00:12:4c:56:a7:8c
ssid=MY_NET
pairwise_cipher=WEP-104
group_cipher=WEP-104
key_mgmt=NONE
wpa_state=COMPLETED
> quit
# dhcp.client -i abc0
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
    inet 127.0.0.1 netmask 0xff000000
abc0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ssid MY_NET nwkey My_Net_Key234
    powersave off
    bssid 00:12:4c:56:a7:8c chan 11
    address: 00:ab:cd:ef:d7:ac
    media: IEEE802.11 autoselect (OFDM54 mode 11g)
    status: active
    inet 10.42.161.233 netmask 0xfffffc00 broadcast 10.42.160.252
#
```

Using a Wireless Access Point (WAP)

A Wireless Access Point (WAP) is a system that allows wireless clients to access the rest of the network or the Internet.

Your WAP will operate in BSS mode. A WAP will have at least one wireless network interface to provide a connection point for your wireless clients, and one wired network interface that connects to the rest of your network. Your WAP will act as a bridge or gateway between the wireless clients, and the wired intranet or Internet.

Creating A WAP

To set up your wireless access point, you first need to start the appropriate driver for your network adapters.

Not all network adapter hardware will support operating as an access point. Refer to the documentation for your specific hardware for further information.

For the wireless access point samples, we'll use the fictitious `devnp-abc.so` driver for the wireless chipsets, and the equally fictitious `devnp-xyz.so` driver for the wired interface. After a default installation, all driver binaries are installed under the directory `$QNX_TARGET/`*cpu*`/lib/dll`.

Use one of the following commands:

- `io-pkt-v4-hc -d abc -d xyz`

  or:

- `io-pkt-v4-hc -d /lib/dll/devnp-abc.so -d /lib/dll/devnp-xyz.so`

  or:

- `io-pkt-v6-hc -d abc -d xyz`

If the driver is installed in a location other than `/lib/dll`, you need to specify the full path and filename of the driver on the command line.

The next step to configure your WAP is to determine whether it will be acting as a gateway or a bridge to the rest of the network, as described below.

Acting as a gateway

When your WAP acts as a gateway, it forwards traffic between two subnets (your wireless network and the wired network).

For TCP/IP, this means that the wireless TCP/IP clients can't directly reach the wired TCP/IP clients without first sending their packets to the gateway (your WAP). Your WAP network interfaces will also each be assigned an IP address.

This type of configuration is common for SOHO (small office, home office) or home use, where the WAP is directly connected to your Internet service provider. Using this type of configuration lets you:

- keep all of your network hosts behind a firewall/NAT
- define and administer your own TCP/IP network

The TCP/IP configuration of a gateway and firewall is the same whether your network interfaces are wired or wireless. For details of how to configure a NAT, visit *http://www.netbsd.org/docs/*.

Once your network is active, you will assign each interface of your WAP an IP address, enable forwarding of IP packets between interfaces, and apply the appropriate firewall and NAT configuration. For more information, see "*DHCP server on WAP acting as a gateway* (p. 59)" in the section on TCP/IP interface configuration.

Acting as a bridge

When your WAP acts as a bridge, it's connecting your wireless and wired networks as if they were one physically connected network (broadcast domain, layer 2). In this case, all the wired and wireless hosts are on the same TCP/IP subnet and can directly exchange TCP/IP packets without the need for the WAP to act as a gateway.

In this case, you don't need to assign your WAP network interfaces an IP address to be able to exchange packets between the wireless and wired networks. A bridged WAP could be used to allow wireless clients onto your corporate or home network and have them configured in the same manner as the wireless hosts. You don't need to add more services (such as DHCP) or manipulate routing tables. The wireless clients use the same network resources that the wired network hosts use.

While it isn't necessary to assign your WAP network interfaces an IP address for TCP/IP connectivity between the wireless clients and wired hosts, you probably will want to assign at least one of your WAP interfaces an IP address so that you can address the device in order to manage it or gather statistics.

To enable your WAP to act as a bridge, you first need to create a bridge interface:

```
ifconfig bridge0 create
```

In this case, `bridge` is the specific interface type, while 0 is a unique instance of the interface type. There can be no space between `bridge` and 0; `bridge0` becomes the new interface name.

Use the `brconfig` command to create a logical link between the interfaces added to the bridge (in this case `bridge0`). This command adds the interfaces `abc0` (our wireless interface) and `wm0` (our wired interface). The up option is required to activate the bridge:

```
brconfig bridge0 add abc0 add wm0 up
```

Remember to mark your bridge as up, or else it won't be activated.

To see the status of your defined bridge interface, you can use this command:

```
brconfig bridge0
```

```
bridge0: flags=41<UP,RUNNING>
    Configuration:
        priority 32768 hellotime 2 fwddelay 15 maxage 20
    Interfaces:
        en0 flags=3<LEARNING, DISCOVER>
            port 3 priority 128
        abc0 flags=3<LEARNING,DISCOVER>
            port 2 priority 128
    Address cache (max cache: 100, timeout: 1200):
```

WEP access point

Enabling WEP network authentication and data encryption is similar to configuring a wireless client, because both the WAP and client require the same configuration parameters.

If you're creating a new wireless network, we recommend you use WPA or WPA2 (RSN) rather than WEP, because WPA and WPA2 provide more better security. You should use WEP only if there are devices on your network that don't support WPA or WPA2.

To use your network adapter as a wireless access point, you must first put the network adapter in host access point mode:

```
ifconfig abc0 mediaopt hostap
```

You will also likely need to adjust the media type (link speed) for your wireless adapter as the auto-selected default may not be suitable. You can view all the available media types with the `ifconfig -m` command. They will be listed in the supported combinations of media type and media options. For example, if the combination of:

```
media OFDM54 mode 11g mediaopt hostap
```

is listed, you could use the command:

```
ifconfig abc0 media OFDM54 mediaopt hostap
```

to set the wireless adapter to use 54 Mbit/s.

The next parameter to specify is the network name or SSID. This can be up to 32 characters long:

```
ifconfig abc0 ssid "my lan"
```

The final configuration parameter is the WEP key. The WEP key must be either 40 bits or 104 bits long. You can either enter 5 or 13 characters for the key, or a 10- to 26-digit hexadecimal value. For example:

```
ifconfig abc0 nwkey corpseckey456
```

You must also mark your network interface as "up" to activate it:

```
ifconfig abc0 up
```

You can also combine all of these commands:

```
ifconfig abc0 ssid "my lan" nwkey corpseckey456 mediaopt hostap up
```

Your network should now be marked as up:

```
ifconfig abc0
```

```
abc0: flags=8943<UP,BROADCAST, RUNNING, PROMISC, SIMPLEX, MULTICAST> mtu 1500
    ssid "my lan" apbridge nwkey corpseckey456
    powersave off
    bssid 11:22:33:44:55:66 chan 2
    address: 11:22:33:44:55:66
    media: IEEE802.11 autoselect hostap (autoselect mode 11b hostap)
    status: active
```

WPA access point

WPA/WPA2 support in the QNX Neutrino RTOS is provided by the `hostapd` daemon.

This daemon is the access point counterpart to the client side `wpa_supplicant` daemon. This daemon manages your wireless network adapter when in access point mode. The `hostapd` configuration is defined in the `/etc/hostapd.conf` configuration file.

Before you start the `hostapd` process, you must put the network adapter into host access point mode:

```
ifconfig abc0 mediaopt hostap
```

You will also likely need to adjust the media type (link speed) for your wireless adapter, as the auto-selected default may not be suitable. You can view all the available media types with the `ifconfig -m` command. They will be listed in the supported combinations of media type and media options. For example, if the combination of:

```
media OFDM54 mode 11g mediaopt hostap
```

is listed, you could use the command:

```
ifconfig abc0 media OFDM54 mediaopt hostap
```

to set the wireless adapter to use 54 Mbit/s.

The remainder of the configuration is handled with the `hostapd` daemon. It automatically sets your network interface as up, so you don't need to do this step with the **ifconfig** utility. Here's a simple **hostapd** configuration file (`/etc/hostapd.conf`):

```
interface=abc0
ssid=my home lan
macaddr_acl=0
auth_algs=1
wpa=1
wpa_passphrase=myhomelanpass23456
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP
```

This configuration uses WPA-PSK for authentication, and AES for data encryption.

> The `auth_algs` and `wpa` are bit fields, not numeric values.

You can now start the **hostapd** utility, specifying the configuration file:

```
hostapd -B /etc/hostapd.conf
```

The **ifconfig** command should show that the network interface is active:

```
ifconfig abc0

abc0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2290
        ssid "my home lan" apbridge nwkey 2:"",0x49e2a9908872e76b3e5e0c32d09b0b52,0x00000000dc710408c04b32b07c9735b0,""
        powersave off
        bssid 00:15:e9:31:f2:5e chan 4
        address: 00:15:e9:31:f2:5e
        media: IEEE802.11 OFDM54 hostap (OFDM54 mode 11g hostap)
        status: active
```

Your WAP should now be available to your clients.

TCP/IP configuration in a wireless network

Client in infrastructure or ad hoc mode

Assigning an IP address to your wireless interface is independent of the 802.11 network configuration and uses the same utilities or daemons as a wired network.

The main issue is whether your TCP/IP configuration is dynamically assigned, or statically configured. A static TCP/IP configuration can be applied regardless of the state of your wireless network connection. The wireless network could be active, or it could be unavailable until later. A dynamically assigned TCP/IP configuration (via the DHCP protocol) requires that the wireless network configuration be active, so that it can reach the DHCP server somewhere on the network. This is typically applied in a network that is centrally administered (using infrastructure mode with a WAP).

The most common usage case is that you're a client using a Wireless Access Point to connect to the network. In this kind of network, there should be a DHCP server available. After the 802.11 network status is active, you just need to start `dhcp.client` to complete your TCP/IP configuration. For example:

```
dhcp.client -iabc0
```

As an alternative, you could use `lsm-autoip.so`. Auto IP is a special case in that it negotiates with its peers on the network as they become available; you don't need to wait until the network link becomes active to launch it. Auto IP will assign your network interface an IP address and resolve any IP address conflicts with your network peers as they're discovered when either your host or the peer changes its current IP address. You will be able to use this IP address once the wireless network is active. For more information, see the documentation for Auto IP.

The last configuration option is a static configuration, which doesn't change without intervention from the user. Here's an example of a static configuration that uses 10.0.0.5 for the wireless interface IP address, and 10.0.0.1 for the network gateway:

```
ifconfig abc0 10.0.0.5
route add default 10.0.0.1

cat /etc/resolv.conf

    domain company.com
    nameserver 10.0.0.2
    nameserver 10.0.0.3
```

The other usage case is an ad hoc network. This network mode is typically made up of a number of standalone peers with no central services. Since there's no central server, it's likely that DHCP services won't be available.

If there are Windows or Apple systems on your ad hoc network, they'll enable the Auto IP protocol to assign an IP address. By using Auto IP, you avoid IP address conflicts (two or more hosts using the same IP address), and you avoid having to configure a new IP address manually. Your IP address will be automatically configured, and you'll be able to exchange TCP/IP packets with your peers.

If you're using a static configuration in an ad hoc network, you'll have the added task of deciding what IP address to use on each system, making sure that there are no conflicts, and that all the IP addresses assigned are on the same subnet, so that the systems can communicate.

DHCP server on WAP acting as a gateway

If you've configured your WAP to act as a gateway, you will have your wireless network on a separate subnet from your wired network. In this case, you could be using infrastructure mode or ad hoc mode.

The instructions below could work in either mode. You'll likely be using infrastructure mode, so that your network is centrally administered. You can implement DHCP services by running `dhcpd` directly on your gateway, or by using `dhcrelay` to contact another DHCP server elsewhere in the ISP or corporate network that manages DHCP services for subnets.

If you're running `dhcpd` on your gateway, it could be that your gateway is for a SOHO. In this case, your gateway is directly connected to the Internet, or to an IP network for which you don't have control or administrative privileges. You may also be using NAT in this case, as you've been given only one IP address by your Internet Service Provider. Alternatively, you may have administrative privileges for your network subnet which you manage.

If you're running `dhcrelay` on your gateway, your network subnet is managed elsewhere. You're simply relaying the DHCP client requests on your subnet to the DHCP server that exists elsewhere on the network. Your relay agent forwards the client requests to the server, and then passes the reply packets back to the client.

These configuration examples assume that you have an interface other than the wireless network adapter that's completely configured to exchange TCP/IP traffic and reach any servers noted in these configurations that exist outside of the wireless network. Your gateway will be forwarding IP traffic between this interface and the wireless interface.

Launching the DHCP server on your gateway

Below is a simple `dhcpd` configuration file, `dhcpd.conf`.

This file includes a subnet range that's dynamically assigned to clients, but also contains two static entries for known servers that are expected to be present at certain IP addresses. One is a printer server, and the other is a network-enabled toaster. The DHCP server configuration isn't specific to wireless networks, and you can apply it to wired networks as well.

```
ddns-update-style none;
```

```
#option subnet-mask 255.255.255.224;
default-lease-time 86400;
#max-lease-time 7200;

subnet 192.168.20.0 netmask 255.255.255.0 {
        range 192.168.20.41 192.168.20.254;
        option broadcast-address 192.168.20.255;
        option routers 192.168.20.1;
        option domain-name-servers 192.168.20.1;
        option domain-name "soho.com";

        host printerserver {
               hardware ethernet 00:50:BA:85:EA:30;
               fixed-address 192.168.20.2;
        }

        host networkenabledtoaster {
               hardware ethernet 00:A0:D2:11:AE:81;
               fixed-address 192.168.20.40;
        }
}
```

The nameserver, router IP, and IP address will be supplied to your wireless network clients. The router IP address is the IP address of the gateway's wireless network interface that's connected to your wireless network. The nameserver is set to the gateway's wireless network adapter, since the gateway is also handling name serving services. The gateway nameserver will redirect requests for unknown hostnames to the ISP nameserver. The internal wireless network has been defined to be 192.168.20.0. Note that we've reserved IP address range 192.168.20.1 through 192.168.20.40 for static IP address assignment; the dynamic range starts at 192.168.20.41.

Now that we have the configuration file, we need to start **dhcpd**.

We need to make sure that the directory `/var/run` exists, as well as `/var/state/dhcp`. The file `/var/state/dhcp/dhcpd.leases` must exist. You can create an empty file for the initial start of the **dhcpd** binary.

When you start **dhcpd**, you must tell it where to find the configuration file if it isn't in the default location. You also need to pass an interface name, as you want only **dhcpd** to service your internal wireless network interface. If we used the fictitious ABC adapter from the wireless discussion, this would be `abc0`:

```
dhcpd -cf /etc/dhcpd.conf abc0
```

Your DHCP server should now be running. If there are any issues, you can start **dhcpd** in a debugging mode using the -d option. The **dhcpd** daemon also logs messages to the system log, `slogger`.

The `dhcrelay` agent doesn't require a configuration file as the DHCP server does; you just need to launch a binary on the command line.

What you must know is the IP address of the DHCP server that's located elsewhere on the network that your gateway is connected to. Once you've launched `dhcrelay`, it forwards requests and responses between the client on your wireless network and the DHCP server located elsewhere on the ISP or corporate network:

```
dhcrelay -i abc0 10.42.42.42
```

In this case, it relays requests from wireless interface (`abc0`), and forwards these requests to the DHCP server 10.42.42.42.

Configuring an access point as a router

To configure an access point as a router:

1. Make sure the outside network interface on your access point is active. That is, make sure your access point is active on the wired network that it's connected to.
2. Configure the access point interface. The simplest mechanism to use for this is WEP.

   Say we want our wireless network to advertise `MY_WIRELESS_NET`, and our WEP secret is `MYWIRELESSWEP`. We have to do the following:

   a. Allow packets coming in from one interface to be forwarded (routed) out another:

   ```
   #sysctl -w net.inet.ip.forwarding=1
   ```

   b. Place the wireless interface into access point mode:

   ```
   #ifconfig in_nic mediaopt hostap
   ```

   c. Configure the wireless interface to be a WEP network with an associated key:

   ```
   #ifconfig in_nic ssid MY_WIRELESS_NET nwkey MYWIRELESSWEP
   ```

   d. Bring up the interface:

   ```
   #ifconfig in_nic 10.42.0.1 up
   ```

3. See above for how you set up DHCP to distribute IP addresses to the wireless client. Briefly, you provide a `dhcpd.conf` with a configuration section as follows, which defines the internal network:

   ```
   subnet 10.42.42.0 netmask 255.255.255.0 {
       range 10.42.0.2 10.42.0.120;
       ...;
   }
   ```

   Then you run `dhcpd`:

   ```
   #dhcpd -cf full_path_to_your_dhcp_config_file -lf \
   full_path_to_your_release_file ni_nic
   ```

   You don't need to specify where your `dhcpd.conf` and release file are if you put them in the default place under `/etc`. For more information, see the entry for `dhcpd` in the *Utilities Reference*.

To use WPA or WPA2, you need to set up and run `hostapd` (the server-side application associated with the client's `wpa_supplicant`) to do the authentication and key exchange for your network.

You can also configure your access point as a NAT network router as follows:

```
#mount -Ttcpip lsm-pfv4.so
```

so that the PF module is loaded, and then use `pfctl` to do the configuration.

For details of how to configure a NAT, visit *http://www.netbsd.org/docs/*.

Chapter 5
Transparent Distributed Processing

Transparent Distributed Processing (also known as Qnet) functions the same under the old `io-net` and new `io-pkt` infrastructures, and the packet format and protocol remain the same. For both `io-net` and `io-pkt`, Qnet is just another protocol (like TCP/IP) that transmits and receives packets.

The Qnet module in Core Networking is now a loadable shared module, `lsm-qnet.so`. We support only the `l4_lw_lite` variant; we no longer support the `qnet-compat` variant that was compatible with QNX Neutrino 6.2.1.

To start the stack with Qnet, type this command:

```
io-pkt-v4 -ddriver -pqnet
```

(assuming you have your ***PATH*** and ***LD_LIBRARY_PATH*** environment variables set up properly). You can also mount the protocol after the stack has started, like this:

```
mount -Tio-pkt full_path_to_dll/lsm-qnet.so
```

Note that `mount` still supports the io-net option, to provide backward compatibility with existing scripts.

The command-line options and general configuration information are the same as they were with `io-net`. For more information, see `lsm-qnet.so` in the *Utilities Reference*.

Using TDP over IP

TDP supports two modes of communications: one directly over Ethernet, and one over IP.

The “straight to Ethernet” L4 layer is faster and more dynamic than the IP layer, but it isn't possible to route TDP packets out of a single layer-2 domain. By using TDP over IP, you can connect to any remote machine over the Internet as follows:

1. TDP must use the DNS resolver to get an IP address from a hostname (i.e., use the resolve=dns option). Configure the local host name and domain, and then make sure that *gethostbyname()* can resolve all the host names that you want to talk to (including the local machine):
   - Use `hostname` to set the host name.
   - Use `setconf` to set the `_CS_DOMAIN` configuration string to indicate your domain.
   - If the hosts aren't in a DNS database, create an appropriate name to host resolution file in `/etc/hosts` which includes the fully qualified node name (including domain) and change the resolver to use the host file instead of using the DNS server (e.g., `setconf` _CS_RESOLVE lookup_file_bind).

     For more information on name resolution, see the description of “Name servers” in the TCP/IP Networking chapter of the QNX Neutrino *User's Guide*.
2. Start (or mount) Qnet with the bind=ip,resolve=dns options. For example:

   ```
   io-pkt-v4-hc -di82544 -pqnet bind=ip,resolve=dns
   ```

   or:

   ```
   mount -Tio-pkt -o bind-ip,resolve=dns full_path_to_dll/lsm-qnet.so
   ```

With raw Ethernet transport, names automatically appear in the `/net` directory. This doesn't happen with TDP over IP; as you perform TDP operations (e.g. `ls /net/host1`), the entries are created as required.

Chapter 6
Network Drivers

Let's take a closer look at the network drivers that you load into `io-pkt`.

Types of network drivers

The networking stack supports the following types of drivers:

- *Native drivers* that are written specifically for the `io-pkt` stack and as such are fully featured, provide high performance, and can run with multiple threads.
- `io-net` drivers that were written for the legacy networking stack `io-net`.
- Ported NetBSD drivers that were taken from the NetBSD source tree and ported to `io-pkt`.

You can tell a native driver from an `io-net` driver by the name:

- `io-net` drivers are named `devn-`*xxxxxx*`.so`
- `io-pkt` native drivers are named `devnp-`*xxxxxx*`.so`

NetBSD drivers aren't as tightly integrated into the overall stack. In the NetBSD operating system, these drivers operate with interrupts disabled and, as such, generally have fewer mutexing issues to deal with on the transmit and receive path. With a straight port of a NetBSD driver, the stack defaults to a single-threaded model, in order to prevent possible transmit and receive synchronization issues with simultaneous execution. If the driver has been carefully analyzed and proper synchronization techniques applied, then a flag can be flipped during the driver attachment, saying that the multi-threaded operation is allowed.

If one driver operates in single-threaded mode, all drivers operate in single-threaded mode.

The native and NetBSD drivers all hook directly into the stack in a similar manner. The `io-net` drivers interface through a “shim” layer that converts the `io-net` binary interface into the compatible `io-pkt` interface. We have a special driver, `devnp-shim.so`, that's automatically loaded when you start an `io-net` driver.

The shim layer provides binary compatibility with existing `io-net` drivers. As such, these drivers are also not as tightly integrated into the stack. Features such as dynamically setting media options or jumbo packets for example aren't supported for these drivers. Given that the driver operates within the `io-net` design context, the drivers won't perform as well as a native one. In addition to the packet receive / transmit device drivers, device drivers are also available that integrate hardware crypto acceleration functionality directly into the stack.

For information about specific drivers, see the *Utilities Reference*:

- `devnp-*` for native `io-pkt` and ported NetBSD drivers. The entry for each driver indicates which type it is.
- `devn-*` for legacy `io-net` drivers

We might not be able to support ported drivers for which the source is publicly available if the vendor doesn't provide documentation to us. While we'll make every effort to help you, we can't guarantee that we'll be able to rectify problems that may occur with these drivers.

Differences between ported NetBSD drivers and native drivers

There's a fine line between native and ported drivers. If you do more than the initial “make it run” port, the feature sets of a ported driver and a native driver aren't really any different.

If you look deeper, there are some differences:

- From a source point of view, a ported driver has a very different layout from a native `io-pkt` driver. The native driver source looks quite similar in terms of content and files to what an `io-net` driver looks like and has all of the source for a particular driver under one directory. The NetBSD driver source is quite different in layout, with source for a particular driver spread out under a specific driver directory, as well as `ic`, `pci`, `usb`, and other directories, depending on the driver type and bus that it's on.
- Ported NetBSD drivers don't allow the stack to run in multi-threaded mode. NetBSD drivers don't have to worry about Rx / Tx threads running simultaneously when run inside of the NetBSD operating system, so there's no need to pay close attention to appropriate locking issues between Rx and Tx.

  For this reason, a configuration flag is, by default, set to indicate that the driver doesn't support multi-threaded access. As a result, the entire stack runs in a single-threaded mode of operation (if one driver can't run in multithreaded mode, no drivers will run with multiple threads). You can change this flag once you've carefully examined the driver to ensure that there are no locking issues.
- NetBSD drivers don't include support for QNX Neutrino-specific utilities, such as `nicinfo`.
- The NetBSD drivers have two different delay functions, both of which take an argument in microseconds. From the NetBSD documentation, *DELAY()* is reentrant (i.e it doesn't modify any global kernel or machine state) and is safe to use in interrupt or process context.

  However, QNX Neutrino's version of *delay()* takes a time in *milliseconds*, so this could result in very long timeouts if used directly as-is in the drivers. We've defined *DELAY()* to do the appropriate conversion of the delay from microseconds to milliseconds, so all NetBSD ported drivers should define *delay()* to be *DELAY()*.

Differences between `io-net` drivers and other drivers

The differences between legacy `io-net` drivers and other drivers include the following:

- The `io-net` drivers export a name space entry, `/dev/io-net/en`*x*. Native drivers don't.

Because of this, a `waitfor` command for such an entry won't work properly in buildfiles or scripts. Use `if_up -p` instead; for example, instead of `waitfor /dev/io-net/en0`, use `if_up -p en0`.

- You can unmount an `io-net` driver (`umount /dev/io-net/en`*x*). With a native driver, you have to destroy it (`ifconfig tsec0 destroy`).
- The `io-net` drivers are all prefixed with `en`. Native drivers have different prefixes for different hardware (e.g. `tsec` for Freescale TSEC devices), although you can override this with the name= driver option (processed by `io-pkt`).
- The `io-net` drivers support the `io-net` *devctl()* commands. Native drivers don't.
- The `io-net` drivers are slower than native drivers, since they use the same threading model as that used in `io-net`.
- The `io-net` driver DLLs are prefixed by `devn-`. Core Networking drivers are prefixed by `devnp-`.
- The `io-net` drivers used the speed and duplex command-line options to override the auto-negotiated link defaults once. Often the use of these options caused more problems than they fixed. Native (and most ported NetBSD drivers) allow their speed and duplex setting to be determined at runtime via a device *ioctl()*, which `ifconfig` uses. See `ifconfig -m` and `ifconfig mediaopt`.

Loading and unloading a driver

You can load drivers into the stack from the command line just as with `io-net`.

For example:

```
io-pkt-v4-hc -di82544
```

This command-line invocation works whether or not the driver is a native driver or an `io-net`-style driver. The stack automatically detects the driver type and loads the `devnp-shim.so` binary if the driver is an `io-net` driver.

Make sure that all drivers are located in a directory that can be resolved by the ***LD_LIBRARY_PATH*** environment variable if you don't want to have to specify the fully qualified name of the device in the command line.

You can also `mount` a driver in the standard way:

```
mount -Tio-pkt /lib/dll/devnp-i82544.so
```

The `mount` command still supports the `io-net` option, to provide backward compatibility with existing scripts:

```
mount -Tio-net /lib/dll/devnp-i82544.so
```

The standard way to remove a driver from the stack is with the `ifconfig` *iface* `destroy` command. For example:

```
ifconfig wm0 destroy
```

Troubleshooting a driver

For native drivers and `io-net` drivers, the `nicinfo` utility is usually the first debug tool that you'll use (aside from `ifconfig`) when problems with networking occur. This will let you know whether or not the driver has properly negotiated at the link layer and whether or not it's sending and receiving packets.

Ensure that the `slogger` daemon is running, and then after the problem occurs, run the `sloginfo` utility to see if the driver has logged any diagnostic information. You can increase the amount of diagnostic information that a driver logs by specifying the verbose command-line option to the driver. Many drivers support various levels of verbosity; you might even try specifying verbose=10.

For ported NetBSD drivers that don't include `nicinfo` capabilities, you can use `netstat -I` *iface* to get very basic packet input / output information. Use `ifconfig` to get the basic device information. Use `ifconfig -v` to get more detailed information.

Problems with shared interrupts

Having different devices sharing a hardware interrupt is kind of a neat idea, but unless you really need to do it—because you've run out of hardware interrupt lines—it generally doesn't help you much. In fact, it can cause you trouble. For example, if your driver doesn't work (e.g., no received packets), check to see if it's sharing an interrupt with another device, and if so, reconfigure your board so it doesn't.

Most of the time, when shared interrupts are configured, there's no good reason for it (i.e., you haven't really run out of interrupts) and this can decrease your performance, because when the interrupt fires, *all* of the devices sharing the interrupt need to run and check to see if it's for them. Some drivers do the "right thing," which is to read registers in their interrupt handlers to see if the interrupt is really for them, and then ignore it if not. But many drivers don't; they schedule their thread-level event handlers to check their hardware, which is inefficient and reduces performance.

If you're using the PCI bus, use the `pci -v` utility to check the interrupt allocation.

Sharing interrupts can vastly increase interrupt latency, depending upon exactly what each of the drivers does. After an interrupt fires, the kernel doesn't reenable it until *all* driver handlers tell the kernel that they've finished handling it. So, if one driver takes a long time servicing a shared interrupt that's masked, then if another device on the same interrupt causes an interrupt during that time period, processing of that interrupt can be delayed for an unknown duration of time.

Interrupt sharing can cause problems, and reduce performance, increase CPU consumption, and seriously increase latency. Unless you really need to do it, don't. If you must share interrupts, make sure your drivers are doing the "right thing."

Writing a new driver

If you're interested in writing a new network driver, see these appendixes in this guide:

- *Writing Network Drivers for `io-pkt`*
- *A Hardware-Independent Sample Driver: `sam.c`*
- *Additional Information*

Debugging a driver using gdb

If you want to use `gdb` to debug a driver, you first have to make sure that your source is compiled with debugging information included.

With your driver code in the correct place in the `sys` tree (`dev_qnx` or `dev`), you can do the following:

```
# cd sys
# make CPULIST=x86 clean
# make CPULIST=x86 CCOPTS=-O0 DEBUG=-g install
```

Now that you have a debug version, you can start `gdb` and set a breakpoint at *main()* in the `io-pkt` binary.

Don't forget to specify your driver in the arguments, and ensure that the ***PATH*** and ***LD_LIBRARY_PATH*** environment variables are properly set up.

After hitting the breakpoint in *main()*, do a `sharedlibrary` command in `gdb`. You should see `libc` loaded in. Set a breakpoint in *dlsym()*. When that's hit, your driver should be loaded in, but `io-pkt` hasn't done the first callout into it. Do a `set solib-search-path` and add the path to your driver, and then do a `sharedlibrary` again. The debugger should load the symbols for your driver, and then you can set a breakpoint where you want your debugging to start.

Dumping 802.11 debugging information

The stack's 802.11 layer can dump debugging information.

You can enable and disable the dumping by using `sysctl` settings. If you do:

```
sysctl -a | grep 80211
```

with a Wi-Fi driver, you'll see `net.link.ieee80211.debug` and `net.link.ieee80211.vap0.debug`. To turn on the debug output, type the following:

```
sysctl -w net.link.ieee80211.debug = 1
sysctl -w net.link.ieee80211.vap0.debug=0xffffffff
```

You can then use `sloginfo` to display the debug log.

Jumbo packets and hardware checksumming

Jumbo packets are packets that carry more payload than the normal 1500 bytes. Even the definition of a jumbo packet is unclear; different people use different lengths.

For jumbo packets to work, the protocol stack, the drivers, and the network switches must all support jumbo packets:

- The `io-pkt` (hardware-independent) stack supports jumbo packets.
- Not all network hardware supports jumbo packets (generally, newer GiGE NICs do).
- Native drivers for `io-pkt` support jumbo packets. For example, `devnp-i82544.so` is a native `io-pkt` driver for PCI, and it supports jumbo packets.

If you can use jumbo packets with `io-pkt`, you can see substantial performance gains because more data can be moved per packet header processing overhead.

To configure a driver to operate with jumbo packets, do this (for example):

```
# ifconfig wm0 ip4csum tcp4csum udp4csum
# ifconfig wm0 mtu 8100
# ifconfig wm0 10.42.110.237
```

For maximum performance, we also turned on hardware packet checksumming (for both transmit and receive) and we've arbitrarily chosen a jumbo packet MTU of 8100 bytes. A little detail: `io-pkt` by default allocates 2 KB clusters for packet buffers. This works well for 1500 byte packets, but for example when an 8 KB jumbo packet is received, we end up with 4 linked clusters. We can improve performance by telling `io-pkt` (when we start it) that we're going to use jumbo packets, like this:

```
# io-pkt-v6-hc -d i82544 -p tcpip pagesize=8192,mclbytes=8192
```

If we pass the pagesize and mclbytes command-line options to the stack, we tell it to allocate contiguous 8 KB buffers (which may end up being two adjacent 4 KB pages, which works fine) for each 8 KB cluster to use for packet buffers. This reduces packet processing overhead, which improves throughput and reduces CPU utilization.

Padding Ethernet packets

If an Ethernet packet is shorter than `ETHERMIN` bytes, padding can be added to the packet to reach the required minimum length. In the interests of performance, the driver software doesn't automatically pad the packets, but leaves it to the hardware to do so if supported. If hardware pads the packets, the contents of the padding depend on the hardware implementation.

Transmit Segmentation Offload (TSO)

Transmit Segmentation Offload (TSO) is a capability provided by some modern NIC cards.

See, for example, *http://en.wikipedia.org/wiki/Large_segment_offload*. Essentially, instead of the stack being responsible for breaking a large IP packet into MTU-sized packets, the driver does it. This greatly offloads the amount of CPU required to transmit large amounts of data.

You can tell if a driver supports TSO by typing `ifconfig` and looking at the capabilities section of the interface output. It will have `tso` marked as one of its capabilities. To configure the driver to use TSO, type (for example):

```
ifconfig wm0 tso4
ifconfig wm0 10.42.110.237
```

Appendix A
Utilities, Managers, and Configuration Files

The utilities, drivers, configuration files, and so on listed below are associated with io-pkt.

For more information, see the *Utilities Reference*.

`brconfig`

Configure network bridge parameters

`hostapd`

Authenticator for IEEE 802.11 networks

`ifconfig`

Configure network interface parameters

`ifwatchd`

Watch for addresses added to or deleted from interfaces and call up/down-scripts for them

`io-pkt`

Network I/O support

`lsm-autoip.so`

AutoIP negotiation module for link-local addresses

`lsm-qnet.so`

Transparent Distributed Processing (native QNX network) module

`nicinfo`

Display information about a network interface controller

`pf`

Packet Filter pseudo-device

`pf.conf`

Configuration file for `pf`

`pfctl`

Control the packet filter (PF) and network address translation (NAT) device

ping

Send ICMP ECHO_REQUEST packets to network hosts (UNIX)

pppoectl

Display or set parameters for a pppoe interface

setkey

Manually manipulate the IPsec SA/SP database

sysctl

Get or set the state of the socket manager

tcpdump

Dump traffic on a network

wpa_cli

WPA command-line client

wpa_passphrase

Set WPA passphrase for a SSID

wpa_supplicant

Wi-Fi Protected Access client and IEEE 802.1X supplicant

For information about drivers, see the devnp-* entries in the *Utilities Reference*.

Appendix B
Writing Network Drivers for `io-pkt`

This appendix is intended to help you understand and write network drivers for io-pkt.

Any network driver can be viewed as the "glue" between the underlying network hardware, and the software infrastructure of `io-pkt`, the operating system protocol stack above it.

So, the "bottom half" of the driver is coded specifically for the particular hardare it supports, and the "top half" of the driver is coded specifically for `io-pkt`.

This document deals specifically with the "top half" of the `io-pkt` driver, which deals with the `io-pkt` software infrastructure.

What does the driver API to `io-pkt` look like?

If you look at an existing `io-pkt` driver, the problem is that it's going to be cluttered up with all sorts of hardware-specific material (i.e., the "bottom half" of driver) which is going to distract you from understanding the API to `io-pkt`.

With this in mind, we've provided a completely hardware-independent sample driver, which can be found in the *`sam.c`* appendix in this guide. For more information about writing a network driver, see the *Additional Information* appendix.

Any driver can be considered to have the following functional areas:

- *Initialization* (p. 81)
- *Interrupt handling and receive packet* (p. 83)
- *Transmit packet* (p. 84)
- *Periodic timers* (p. 85)
- *Link status* (p. 86)
- *Out-of-band control* (p. 86)
- *Shutdown* (p. 87)

This appendix also covers the following advanced topics:

- *Delays* (p. 88)
- *Threading* (p. 89)

Let's take a look at each functional area.

Initialization

Initialization is probably the trickiest part of an `io-pkt` driver because part of the initialization code will be called over and over again by `io-pkt`, so you must code it accordingly. It's very easy to have a driver that works at first, but stops working after `io-pkt` reinitializes it.

Initialization begins with this:

```
struct nw_dll_syms sam_syms[] = {
        {"iopkt_drvr_entry", &IOPKT_DRVR_ENTRY_SYM(sam)},
        {NULL, NULL}
};
```

This tells `io-pkt` to execute the *sam_entry()* function, which in turn calls the *dev_attach()* function for every instance of the hardware, of which there may be none, one, or several.

The *dev_attach()* function, through preprocessor trickery, gets a pointer to the following, via the &*sam_ca* parameter:

```
CFATTACH_DECL(sam,
    sizeof(struct sam_dev),
    NULL,
    sam_attach,
    sam_detach,
    NULL);
```

So for each instance of the hardware, the *sam_attach()* function will be called once and only once. The *sam_attach()* function basically does two things: allocate resources (e.g., those required for the hardware) and hook itself up to `io-pkt`.

Looking at *sam_attach()* we can see it hooking itself up to `io-pkt` in two main ways:

- One is by setting the callout functions in its `ifp` structure. For example, when `io-pkt` wants to transmit a packet, it calls the *ifp->if_start* function pointer, which has nothing to do with initialization, by the way. In *sam_attach()*, we see that the *ifp->if_start* function pointer is set to the address of the *sam_start()* packet transmit function.
- Second is by setting up for the hardware interrupt by calling the *interrupt_entry_init()* `io-pkt` function, which is passed as a parameter, a pointer to the `sc_inter` structure in the per-instance device structure.

  The `sc_inter` struct contains pointers to the *sam_process_interrupt()* and *sam_enable_interrupt()* functions, and also the per-instance device structure pointer (*sam*).

Note that *pthread_create() isn't* called. This is an important detail about the threading model of `io-pkt` drivers: whenever the driver wishes to execute, it must do so under control of (i.e., be called by) `io-pkt`. This quite specifically includes asynchronous events such as hardware interrupts (as discussed above) and also periodic timers via the *callout_msec()* `io-pkt` function.

This completes the part of the driver initialization that's called once. Note that the network hardware won't function at this point; no packets will be received (or transmitted) until someone executes the `ifconfig` utility. For example:

```
ifconfig sam0 10.42.107.238
```

Now, `io-pkt` will call the *ifp->if_init* function pointer for the sample driver, which in the attach function was set to be *sam_init()*. This is where the hardware would be enabled.

Remember, the *ifp->if_init* function can and will be called over and over again by `io-pkt`. For example, if someone does this:

```
ifconfig sam0 mtu 8100
```

then the *ifp->if_init* function in the driver will be called again by `io-pkt`. So, it's up to the driver to initialize the hardware as specified.

We can clearly see from this example that it would be an error of the driver to set the MTU in the attach function. Generally the init function should audit the current hardware configuration and correct it to match the new configuration. It would be a mistake to disable the hardware and initialize it all over again as a small change would then interrupt any current traffic flows.

Summary: the attach function is called once, to allocate resources and to hook up to `io-pkt`. The init function is called over and over again, to configure and enable the hardware.

It's worth mentioning that if you wish to write a driver for a PCI NIC, there's a little dance you need to go through for vendor and device ID tables and scanning. Of course, since `sam.c` was written to be a hardware-independent example, it doesn't have any of that code in it. The `devnp-e1000.so` driver includes this as well as checking the capabilities for using MSI or MSI-X. Similar concerns apply to a USB NIC, and the `devnp-asix.so` driver is an example.

Interrupt handling & receive packet

You'll note that there are two different *sam_isr()* functions provided. The easiest way to handle an interrupt is to simply use the kernel *InterruptMask()* function. A slightly more complicated way to handle the interrupt is to write to a hardware register to mask the interrupt, which works better if the interrupt is being shared with another device, and might be just a little bit faster.

Either way, the *sam_isr()* function needs to mask the interrupt and queue the appropriate function to perform the interrupt work by calling *interrupt_queue()*. In the case of multiple hardware functions sharing the same interrupt, it's common to have multiple process interrupt functions, and determine in the ISR which one to enqueue.

Once the ISR completes, the return value from *interrupt_queue()* causes `io-pkt` to wake up, and calls the driver's *sam_process_interrupt()* function via the *sam->sc_inter.func* function pointer.

The *sam_process_interrupt()* function will do whatever the hardware requires: perhaps reading count registers, error handling, etc. It might or might not service the transmit side of the hardware (generally not recommended because of the negative performance impact of enabling the transmit complete interrupt, but see below).

It will however service the receive side of the hardware: any filled received packet are drained from the hardware, new empty packets are passed down to the hardware, and the filled received packets are passed up to `io-pkt` using the *ifp->if_input* function pointer.

A return value of 0 implies that the interrupt processing function has returned without completing all of its work. This will permit other interfaces to run their interrupt processing by placing *sam_process_interrupt()* at the end of the run queue.

Once *sam_process_interrupt()* completes all its processing and returns 1, then *sam_enable_interrupt()* will be called to enable the interrupts once more.

Transmit packet

As noted above, when `io-pkt` wishes to transmit a packet, it will call the driver's *ifp->if_start* function pointer, which was set to *sam_start()* in the attach function.

Generally the first thing you do here is see if you have the hardware resources (descriptors, buffers, whatever) available to transmit a packet. If the hardware runs out of transmit resources, it should return from the *ifp->if_start* function, leaving `IFF_OACTIVE` set:

```
ifp->if_flags_tx |= IFF_OACTIVE;
```

but remember to release the transmit mutex as described below!

With this flag set, `io-pkt` will no longer call the *ifp->if_start* function when adding a packet to the output queue of the interface. At this point, it'is up to the driver to detect when the out-of-resources condition has been cleared (either through periodic retries or through some other notification such as transmit completion interrupts). The driver should then acquire the transmit mutex and call the start function again to transmit the data in the output queue.

What most drivers do is loop in the *ifp->if_start* function, passing packets down to the hardware until there aren't any more packets to be transmitted, or the hardware resources aren't available to permit packet loading for transmission, whichever comes first.

There are a couple of handy macros that you can use here:

- You can use the *IFQ_POLL()* macro to peek at the transmit queue and see if there are any more packets from `io-pkt` ready for transmitting. If there are none, you're done.

- You use the *IFQ_DEQUEUE()* macro to unlink the first queued packet from the transmit queue. Some drivers just use this function, and don't bother with the *IFQ_POLL()* macro. But once you've dequeued the packet, you must transmit it!

This really isn't very complicated. The main thing to remember is that before you return from this function, you must release the transmit mutex as follows:

```
NW_SIGUNLOCK_P(&ifp->if_snd_ex, iopkt_selfp, wtp);
```

Note that the sample driver, in the start function, calls `m_free(m)` to release the transmitted packet. It does this to avoid a memory leak, but you probably don't want to do that if you have a descriptor-based NIC.

If you have a NIC that unfortunately requires that you copy the transmit packet into a buffer, then you should immediately call `m_free(m)`, which tells `io-pkt` that the buffer is available for reuse, and it will be written to.

However, if you have a descriptor-based NIC, you *don't* copy the transmitted packet: the hardware does the DMA, and you want to release the packet buffer only after the DMA has completed sometime later, to avoid this packet from being overwritten.

If you look at most driver source, any descriptor-based NIC will have a “harvest” or “reap” function that will check for transmitted descriptors, and will at that point release the transmit packet buffer.

This requires that you squirrel away a pointer to the transmit packet (mbuf) somewhere. Often hardware will have a few bytes free in the descriptor for this purpose, or if not, you must maintain a corresponding array of mbufs which you index into while harvesting descriptors.

Note that packets typically come down as multiple buffers e.g., typically for TCP 3 buffers, first containing the headers, second containing the remnants of the previous mbuf and the third containing the start of the next mbuf. Badly fragmented packets may require copying in to a new contiguous buffer depending on the capabilities of the hardware and the degree of buffer fragmentation. This will obviously have a performance impact, so you should avoid it where possible.

Periodic timers

Network drivers frequently need periodic timers to perform such housekeeping functions as link maintenance and transmit descriptor harvesting. An `io-pkt` driver *shouldn't* create its own thread or asynchronous timer via an OS function. The way you set up a periodic timer is as follows in the *ifp->if_init* function:

```
callout_msec(&dev->mii_callout, 2 * 1000, dev_monitor, dev);
```

This will cause the dev_monitor() function to be called by an `io-pkt` thread after two seconds have elapsed.

The gotcha is that at the end of the *dev_monitor()* function, it must rearm its periodic timer call by making the above call again. It's a one-shot—not a repetitive—timer. You may need to add a "run_timer" variable and clear it as well as calling *callout_stop()* when stopping the timer, and only call *callout_msec()* at the end of the *dev_monitor()* function if this variable isn't set. This will close the window on a race condition where the *dev_monitor()* function has started running but not completed when another thread does a *callout_stop()*, then at the completion of the *dev_monitor()* function *callout_msec()* is called again restarting the timer that's supposed to be stopped.

You should create timers only once with a call to *callout_init()*:

```
callout_init (&dev->mii_callout);
```

They can have *callout_msec()* called multiple times, and it will start a stopped timer or reset a currently running timer. Calling *callout_stop()* on a stopped timer will not cause any issues, but calling *callout_init()* more than once will break things. Typically the *callout_init()* will happen in the *ifp->if_attach()* function, which is only called once per device, while *callout_msec()* will happen in the *ifp->if_init()* and also the callback itself; because it resets a running timer and starts a stopped one, there's no need for any further locking. The callout will typically be stopped via a call to *callout_stop()* in the *ifp->if_stop()* function.

If you call into the transmit code to harvest descriptors, you should lock the transmit mutex to avoid corrupting your data and registers, by using the *NW_SIGLOCK()* macro.

Link status events

Users should be notified about link layer state changes. This is done via the *if_link_state_change()* function:

```
if_link_state_change(ifp, LINK_STATE_UP);
if_link_state_change(ifp, LINK_STATE_DOWN);
```

Out of band (control)

Out-of-band (non-data) control of the driver is accomplished by the *ifp->if_ioctl* function pointer which is set to *sam_ioctl()* in the attach function.

The ioctl function can be very simple (empty) or quite complex, depending upon the features supported. For backward compatibility of the `nicinfo` utility (for example, `nicinfo sam0`), you might wish to add support for the `SIOCGDRVCOM DRVCOM_CONFIG`/`DRVCOM_STATS` commands.

If your driver supports hardware checksumming, you probably want to support the `SIOCSIFCAP` command (see examples).

If you want your driver to display its media link speed and duplex via the ifconfig utility:

```
ifconfig -v
```

you want to add support for the `SIOCGIFMEDIA` and `SIOCSIFMEDIA` commands, which actually allow the media speed and duplex to be set via the `ifconfig` utility. Run this:

```
ifconfig -m
```

The `io-pkt` drivers that support the setting of media link speed and duplex via ifconfig will have a source file called `bsd_media.c`. Typically this file is similar across many drivers; they all interface to `io-pkt` quite similarly, and only minor hardware-specific differences exist.

Finally, the `ioctl` interface is how the multicast receive addresses are enabled. See `sam.c` for examples on how these addresses are obtained from `io-pkt`; the *ETHER_FIRST_MULTI()* and *ETHER_NEXT_MULTI()* macros are used for this.

Shutdown

The shutdown scenarios are:

An `ifconfig sam0 down` command calls *sam_stop()*.

This should stop any transmitting and receiving and clear any in-use buffers so stale traffic doesn't appear when bringing the interface back up. Note that while buffers should be cleaned up and Tx/Rx stopped, the rest of the driver structures and hardware should be left intact. The next call in to the driver will likely be triggered by an `ifconfig up` command, which will call *sam_init()* and have everything up and running again.

An `ifconfig sam0 destroy` command calls *sam_detach()*.

This should reset the hardware and clear up all memory. The driver is about to be unmounted from `io-pkt`, leaving `io-pkt` still running. A suggested testcase is to loop around mounting the driver, `ifconfig` it with an address, run some traffic, then `ifconfig destroy` the interface to unmount the driver. It should be possible to do this multiple times with no memory leaks.

An `io-pkt` exit or crash calls *sam_shutdown()*.

This should simply reset the hardware to stop any DMA. Any further cleanup of buffers or structures should be avoided, as it could cause a further crash (masking the original root cause in the core file) as memory is potentially corrupted.

The stop function is specified as one of the standard callbacks, while the detach function is part of the same preprocessor trickery that specified the attach function:

```
CFATTACH_DECL(sam,
    sizeof(struct sam_dev),
    NULL,
    sam_attach,
    sam_detach,
    NULL);
```

The *sam_shutdown()* is specified a little differently:

```
sam->sc_sdhook = shutdownhook_establish(sam_shutdown, sam);
```

It's important to remember to set this in the attach function and equally to clear it in the detach function with:

```
shutdownhook_disestablish(sam->sc_sdhook);
```

Delays

When talking to hardware, a driver often needs to delay for a short time. Recall that in an **io-pkt** driver, all functions are called from the **io-pkt** threads, and not from driver threads. This can lead to issues when there are multiple interfaces, and a delay in the driver on one interface impacts data flow on another.

Internally **io-pkt** uses a pseudo-threading method to avoid blocking, and in certain circumstances we can make use of this in the driver. The one scenario in which it is impossible to delay is a timer callback (see "*Periodic Timers* (p. 85)") where the only possible way to delay would be to set a new timer. Also at **io-pkt** startup, the pseudo-threading mechanism is not yet initialized, so it can't be used, however because everything is starting up, it's acceptable to use a standard delay mechanism.

Here's an example of a 0.5 second delay:

```
if (!ISSTART && ISSTACK) {
    /*
     * Called from an io-pkt thread and not at startup so can't
     * use normal delay, work out what type of delay to use.
     */
    if (curproc == stk_ctl.proc0) {
 /*
  * Called from a callout, can only do another callout.
  * If ltsleep is tried it returns success without
  * actually sleeping.
  */
 callout_msec(&dev->delay_callout, 500, next_part, dev);
 return;
    }
    /*
     * Normal io-pkt thread case. Use ltsleep to avoid blocking
     * other interfaces
     */
    timo = hz / 2;
    ltsleep(&wait, 0, "delay", timo, NULL);
} else {
    /*
```

```
     * Either io-pkt is starting up or called from a different
     * thread so will not block other interfaces. Just use delay.
     */
    delay(500);
}
```

Threading

Earlier we mentioned that a driver shouldn't create its own threads and should run under the `io-pkt` threads. There are some rare situations where a driver needs a thread to handle some other aspect of the hardware (e.g., a USB or SDIO interaction), but in general extra threads should be avoided. If you're in the unlikely scenario of needing a thread, then there are some extra steps that need to be taken with threads in `io-pkt`. While it's possible to create standard threads via *pthread_create()*, this isn't recommended, as they must not have anything to do with mbufs or call back in to `io-pkt` functions.

In `io-pkt`, mbuf handling threads are created by *nw_pthread_create()* rather than *pthread_create()*:

```
nw_pthread_create(&tid, NULL, thread_fn, dev, 0,
    thread_init_fn, dev);
```

The additional thread initialization function must at a minimum set the threads name to differentiate it from the standard `io-pkt` threads, and also set up the quiesce handler. It's permissible to perform other initializations, but at a minimum you must set up the name and the quiesce handler:

```
static int thread_init_fn (void *arg)
{
    struct nw_work_thread *wtp;
    dev_handle_t  *dev = (dev_handle_t *)arg;

    pthread_setname_np(0, "My driver thread");

    wtp = WTP;

    wtp->quiesce_callout = thread_quiesce;
    wtp->quiesce_arg = dev;

    return EOK;
}
```

The thread name should easily identify which driver it's associated with, and if there are multiple threads, then also the threads' purpose. As an example:

```
# pidin -p io-pkt-v4 thread
     pid name                 thread name          STATE       Blocked
    4100 sbin/io-pkt-v4     io-pkt main          SIGWAITINFO
    4100 sbin/io-pkt-v4     io-pkt#0x00          RECEIVE     1
    4100 sbin/io-pkt-v4     asixx Rx             RECEIVE     22
```

The threads in this example are:

io-pkt main

Used for the signal handler and also to handle any blockop requests.

io-pkt#0x00

A thread created by io-pkt for the main io-pkt work. Further numbered threads are created by io-pkt in the case of additional CPUs and additional *interrupt_entry_init()* calls.

dm814x Rx

An example of a driver thread, in this case from the devnp-asix.so driver and used in the Rx processing to handle special low-latency packets when io-pkt is busy servicing other requests. Failure to provide a name will result in the thread's being named as an additional io-pkt numbered thread, resulting in confusion between what is an io-pkt processing thread and what is a driver thread.

The quiesce function is called in two scenarios:

- The first is if io-pkt needs to alter certain structures (such as further *nw_pthread_create()* calls), it requires all the other threads to block while the structures are updated.
- The second case is when the threads are being killed off, for example, at io-pkt's exit time.

The *die* parameter is used to differentiate between the two scenarios. Note that the quiesce function is actually called from an io-pkt thread and needs to notify the driver thread to call *quiesce_block()* through (for example) global variables or a message pulse. Here's an example where the thread is looping around continuously, so global variables can be used:

```
static int quiescing = 0;
static int quiesce_die = 0;

static void thread_quiesce (void *arg, int die)
{
    dev_handle_t *dev = (dev_handle_t *)arg;

    quiescing = 1;
    quiesce_die = die;
}

static void *thread_fn (void *arg)
{

    while (1) {
 if (quiescing) {
     if (quiesce_die) {
  /*
   * Thread will terminate on calling
   * quiesce_block(), clean up here
   * if required.
   */
     }
     quiesce_block(quiesce_die);
```

```
            quiescing = 0;
        }

        /* Do normal thread work */
        }
    }
}
```

Appendix C
A Hardware-Independent Sample Driver: `sam.c`

This is the sample code that's discussed in the *Writing Network Drivers for* ***`io-pkt`*** appendix.

```
/*
 * $QNXtpLicenseC:
 * Copyright 2007, 2014, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#include <io-pkt/iopkt_driver.h>
#include <sys/io-pkt.h>
#include <sys/syspage.h>
#include <sys/device.h>
#include <device_qnx.h>
#include <net/if_ether.h>
#include <net/if_media.h>
#include <net/netbyte.h>
#include <net80211/ieee80211_var.h>

int sam_entry(void *dll_hdl, struct _iopkt_self *iopkt, char *options);

int sam_init(struct ifnet *);
void sam_stop(struct ifnet *, int);

void sam_start(struct ifnet *);
int sam_ioctl(struct ifnet *, unsigned long, caddr_t);

const struct sigevent * sam_isr(void *, int);
int sam_process_interrupt(void *, struct nw_work_thread *);
int sam_enable_interrupt(void *);

void sam_shutdown(void *);


struct _iopkt_drvr_entry IOPKT_DRVR_ENTRY_SYM(sam) = IOPKT_DRVR_ENTRY_SYM_INIT(sam_entry);

#ifdef VARIANT_a
#include <nw_dl.h>
/* This is what gets specified in the stack's dl.c */
struct nw_dll_syms sam_syms[] = {
        {"iopkt_drvr_entry", &IOPKT_DRVR_ENTRY_SYM(sam)},
        {NULL, NULL}
};
#endif

const uint8_t etherbroadcastaddr[ETHER_ADDR_LEN] =
    { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };

struct sam_dev {
 struct device  sc_dev; /* common device */
 struct ethercom  sc_ec; /* common ethernet */
 nic_config_t  cfg; /* nic information */
 struct ieee80211com sc_ic; /* common 80211 */
 /* whatever else you need follows */
 struct _iopkt_self *sc_iopkt;
 int   sc_iid;
 int   sc_irq;
 int   sc_intr_cnt;
```

```
 int   sc_intr_spurious;
 struct _iopkt_inter sc_inter;
 void   *sc_sdhook;
};

int sam_attach(struct device *, struct device *, void *);
int sam_detach(struct device *, int);

CFATTACH_DECL(sam,
 sizeof(struct sam_dev),
 NULL,
 sam_attach,
 sam_detach,
 NULL);

/*
 * Initial driver entry point.
 */
int
sam_entry(void *dll_hdl,  struct _iopkt_self *iopkt, char *options)
{
 int  instance, single;
 struct device *dev;
 void  *attach_args;

 /* parse options */

 /* do options imply single? */
 single = 1;

 /* initialize to whatever you want to pass to sam_attach() */
 attach_args = NULL;

 for (instance = 0;;) {
  /* Apply detection criteria */

  /* Found one */
  dev = NULL; /* No Parent */
  if (dev_attach("sam", options, &sam_ca, attach_args,
      &single, &dev, NULL) != EOK) {
   break;
  }
  dev->dv_dll_hdl = dll_hdl;
  instance++;


  if (/* done_detection || */ single)
   break;
 }

 if (instance > 0)
  return EOK;

 return ENODEV;
}

int
sam_attach(struct device *parent, struct device *self, void *aux)
{
 int   err;
 struct sam_dev  *sam;
 struct ifnet  *ifp;
 uint8_t   enaddr[ETHER_ADDR_LEN];
 struct qtime_entry *qtp;

 /* initialization and attach */

 sam = (struct sam_dev *)self;
 ifp = &sam->sc_ec.ec_if;

 sam->sc_iopkt = iopkt_selfp;

 /*
  * CAUTION: As an example we attach to the system timer interrupt.
  * This would be the network hardware interrupt in a real
  * driver. When this sample driver is run it masks and unmasks
  * the system timer interrupt in io-pkt. This may cause problems
  * with other timer calls in other drivers, potentially even
  * leading to a deadlock. It is safe to run by itself in io-pkt.
  */
```



```
 qtp = SYSPAGE_ENTRY(qtime);
 sam->sc_irq = qtp->intr;

 if ((err = interrupt_entry_init(&sam->sc_inter, 0, NULL,
     IRUPT_PRIO_DEFAULT)) != EOK)
  return err;

 sam->sc_inter.func   = sam_process_interrupt;
 sam->sc_inter.enable = sam_enable_interrupt;
 sam->sc_inter.arg    = sam;

 sam->sc_iid = -1; /* not attached yet */

 /* set capabilities */
#if 0
 ifp->if_capabilities_rx = IFCAP_CSUM_IPv4 | IFCAP_CSUM_TCPv4 | IFCAP_CSUM_UDPv4;
 ifp->if_capabilities_tx = IFCAP_CSUM_IPv4 | IFCAP_CSUM_TCPv4 | IFCAP_CSUM_UDPv4;

 sam->sc_ec.ec_capabilities |= ETHERCAP_JUMBO_MTU;
#endif

 ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;

 /* Set callouts */
 ifp->if_ioctl = sam_ioctl;
 ifp->if_start = sam_start;
 ifp->if_init = sam_init;
 ifp->if_stop = sam_stop;
 IFQ_SET_READY(&ifp->if_snd);

 ifp->if_softc = sam;

 /* More callouts for 80211... */

 strcpy(ifp->if_xname, sam->sc_dev.dv_xname);
 if_attach(ifp);

 {
  int i;
  for (i = 0; i < ETHER_ADDR_LEN; i++)
   enaddr[i] = i;
 }
#if 1
 /* Normal ethernet */
 ether_ifattach(ifp, enaddr);
#else
 /* 80211 */
 memcpy(sam->sc_ic.ic_myaddr, enaddr, ETHER_ADDR_LEN);
 ieee80211_ifattach(&sam->sc_ic);
#endif
 sam->sc_sdhook = shutdownhook_establish(sam_shutdown, sam);

 return EOK;
}

void sam_set_multicast(struct sam_dev *sam)
{
 struct ethercom   *ec = &sam->sc_ec;
 struct ifnet   *ifp = &ec->ec_if;
 struct ether_multi  *enm;
 struct ether_multistep  step;

 ifp->if_flags &= ~IFF_ALLMULTI;

 ETHER_FIRST_MULTI(step, ec, enm);
 while (enm != NULL) {
               if (memcmp(enm->enm_addrlo, enm->enm_addrhi, ETHER_ADDR_LEN)) {
                       /*
                        * We must listen to a range of multicast addresses.
                        * For now, just accept all multicasts, rather than
                        * trying to filter out the range.
                        * At this time, the only use of address ranges is
                        * for IP multicast routing.
                        */
                       ifp->if_flags |= IFF_ALLMULTI;
   break;
               }
  /* Single address */
  printf("Add %2x:%2x:%2x:%2x:%2x:%2x to mcast filter\n",
         enm->enm_addrlo[0],enm->enm_addrlo[1],
```

```
        enm->enm_addrlo[0],enm->enm_addrlo[1],
        enm->enm_addrlo[0],enm->enm_addrlo[1]);
 }

 if ((ifp->if_flags & IFF_ALLMULTI) != 0) {
  printf("Enable multicast promiscuous\n");
 } else {
  printf("Disable multicast promiscuous\n");
 }
}

int
sam_init(struct ifnet *ifp)
{
 int  ret;
 struct sam_dev *sam;

 /*
  * - enable hardware.
  *   - look at ifp->if_capenable_[rx/tx]
  *   - enable promiscuous / multicast filter.
  * - attach to interrupt.
  */

 sam = ifp->if_softc;

 if(memcmp(sam->cfg.current_address, LLADDR(ifp->if_sadl), ifp->if_addrlen)) {
  memcpy(sam->cfg.current_address, LLADDR(ifp->if_sadl), ifp->if_addrlen);
  /* update the hardware */
 }

 if (sam->sc_iid == -1) {
  if ((ret = InterruptAttach_r(sam->sc_irq, sam_isr,
      sam, sizeof(*sam), _NTO_INTR_FLAGS_TRK_MSK)) < 0) {
   return -ret;
  }
  sam->sc_iid = ret;
 }

 sam_set_multicast(sam);
 ifp->if_flags |= IFF_RUNNING;

 return EOK;
}

void
sam_stop(struct ifnet *ifp, int disable)
{
 struct sam_dev *sam;

 /*
  * - Cancel any pending io
  * - Clear any interrupt source registers
  * - Clear any interrupt pending registers
  * - Release any queued transmit buffers.
  */

 sam = ifp->if_softc;

 if (disable) {
  if (sam->sc_iid != -1) {
   InterruptDetach(sam->sc_iid);
   sam->sc_iid = -1;
  }
  /* rxdrain */
 }

 ifp->if_flags &= ~IFF_RUNNING;
}

void
sam_start(struct ifnet *ifp)
{
 struct sam_dev  *sam;
 struct mbuf  *m;
 struct nw_work_thread *wtp;

 sam = ifp->if_softc;
 wtp = WTP;
```

```
 for (;;) {
  IFQ_POLL(&ifp->if_snd, m);
  if (m == NULL)
   break;

  /*
   * Can look at m to see if you have the resources
   * to transmit it.
   */

  IFQ_DEQUEUE(&ifp->if_snd, m);
  /* You're now committed to transmitting it */
  if (sam->cfg.verbose) {
   printf("Packet sent\n");
  }
  m_freem(m);

  ifp->if_opackets++;  // for ifconfig -v
  // or if error:  ifp->if_oerrors++;
 }

 NW_SIGUNLOCK_P(&ifp->if_snd_ex, iopkt_selfp, wtp);
}

int
sam_ioctl(struct ifnet *ifp, unsigned long cmd, caddr_t data)
{
 struct sam_dev *sam;
 int  error;

 sam = ifp->if_softc;
 error = 0;

 switch (cmd) {
 default:
  error = ether_ioctl(ifp, cmd, data);
  if (error == ENETRESET) {
   /*
    * Multicast list has changed; set the
    * hardware filter accordingly.
    */
   if ((ifp->if_flags & IFF_RUNNING) == 0) {
    /*
     * Interface is currently down: sam_init()
     * will call sam_set_multicast() so
     * nothing to do
     */
   } else {
    /*
     * interface is up, recalculate and
     * reprogram the hardware.
     */
    sam_set_multicast(sam);
   }
   error = 0;
  }
  break;
 }

 return error;
}

int
sam_detach(struct device *dev, int flags)
{
 struct sam_dev *sam;
 struct ifnet *ifp;

 /*
  * Clean up everything.
  *
  * The interface is going away but io-pkt is staying up.
  */
 sam = (struct sam_dev *)dev;
 ifp = &sam->sc_ec.ec_if;

 sam_stop(ifp, 1);
#if 1
 ether_ifdetach(ifp);
```

```
#else
 ieee80211_ifdetach(&sam->sc_ic);
#endif

 if_detach(ifp);

 shutdownhook_disestablish(sam->sc_sdhook);

 return EOK;
}

void
sam_shutdown(void *arg)
{
 struct sam_dev *sam;

 /* All of io-pkt is going away.  Just quiet hardware. */

 sam = arg;

 sam_stop(&sam->sc_ec.ec_if, 1);
}

#ifndef HW_MASK
const struct sigevent *
sam_isr(void *arg, int iid)
{
 struct sam_dev  *sam;
 struct _iopkt_inter *ient;

 sam = arg;
 ient = &sam->sc_inter;

 /*
  * Close window where this is referenced in sam_enable_interrupt().
  * We may get an interrupt, return a sigevent and have another
  * thread start processing on SMP before the InterruptAttach()
  * has returned.
  */
 sam->sc_iid = iid;

 InterruptMask(sam->sc_irq, iid);

 return interrupt_queue(sam->sc_iopkt, ient);
}
#else
const struct sigevent *
sam_isr(void *arg, int iid)
{
 struct sam_dev  *sam;
 struct _iopkt_self *iopkt;
 const struct sigevent *evp;
 struct inter_thread *itp;

 sam = arg;
 iopkt = sam->sc_iopkt;
 evp = NULL;

#ifdef READ_CAUSE_IN_ISR
 /*
  * Trade offs.
  * - Doing this here means another register read across the bus.
  * - If not sharing interrupts, this boils down to exactly the
  *   same amount of work but doing more of it in the isr.
  * - If sharing interupts, can short circuit some work in the
  *   stack here.
  * - Maybe trade off is to only do it if we're detecting
  *   spurious interrupts which should happen under heavy
  *   shared interrupt load?
  */
#ifdef READ_CAUSE_ONLY_ON_SPURIOUS
 if (ient->spurrious) {
#endif
  if (ient->on_list == 0 &&
      (sam->sc_intr_cause = i82544->reg[I82544_ICR]) == 0) {
   return NULL; /* Not ours */
  }
  sam->sc_flag |= CAUSE_VALID;
#ifdef READ_CAUSE_ONLY_ON_SPURIOUS
 }
```

```
#endif
#endif

 /*
  * We have to make sure the interrupt is masked regardless
  * of our on_list status.  This is because of a window where
  * a shared (spurious) interrupt comes after on_list
  * is knocked down but before the enable() callout is made.
  * If enable() then happened to run after we masked, we
  * could end up on the list without the interrupt masked
  * which would cause the kernel more than a little grief
  * if one of our real interrupts then came in.
  *
  * This window doesn't exist when using kermask since the
  * interrupt isn't unmasked until all the enable()s run
  * (mask count is tracked by kernel).
  */

 /*
  * If this was controling real hardware, mask of
  * interrupts here. eg from i82544 driver:
  */
 i82544->reg[I82544_IMC] = 0xffffffff;

 return interrupt_queue(sam->sc_iopkt, ient);
}
#endif
int
sam_process_interrupt(void *arg, struct nw_work_thread *wtp)
{
 struct sam_dev  *sam;
 struct mbuf   *m;
 struct ifnet  *ifp;
 struct ether_header *eh;

 sam = arg;
 ifp = &sam->sc_ec.ec_if;


 if ((sam->sc_intr_cnt++ % 1000) == 0) {
  /* Send a packet up */
  m = m_getcl_wtp(M_DONTWAIT, MT_DATA, M_PKTHDR, wtp);

  if (!m) {
            ifp->if_ierrors++;  // for ifconfig -v
   return 1;
  }

  m->m_pkthdr.len = m->m_len = sizeof(*eh);

  // ip_input() needs this
  m->m_pkthdr.rcvif = ifp;

  // dummy up a broadcasted IP packet for testing
  eh = mtod(m, struct ether_header *);
  eh->ether_type = ntohs(ETHERTYPE_IP);
  memcpy(eh->ether_dhost, etherbroadcastaddr, ETHER_ADDR_LEN);

  ifp->if_ipackets++; // for ifconfig -v

  (*ifp->if_input)(ifp, m);

  printf("sam_process_interrupt %d\n", sam->sc_intr_cnt);
 }

 /*
  * return of 1 means were done.
  *
  * If we notice we're taking a long time (eg. processed
  * half our rx descriptors) we could early out with a
  * return of 0 which lets other interrupts be processed
  * without calling our interrupt_enable func.  This
  * func will be called again later.
  */
 return 1;
}
#ifndef HW_MASK
int
sam_enable_interrupt(void *arg)
{
```

```
 struct sam_dev *sam;

 sam = arg;
 InterruptUnmask(sam->sc_irq, sam->sc_iid);

 return 1;
}
#else
int
sam_enable_interrupt(void *arg)
{
 struct sam_dev *sam;

 sam = arg;
 /* eg from i82544 driver */

 i82544->reg[I82544_IMS] = i82544->intrmask;

 return 1;
}
#endif



#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix D
Additional information

The `io-pkt` utility (`io-pkt-v4`, `io-pkt-v4-hc`, and `io-pkt-v6-hc`) is the Neutrino network manager. This is a process outside of the kernel space and executes in the application space. Along with providing the network manager framework, it also includes TCP/IP (TCP, UDP, IPv4, IPv6) support, along with PPP services and a variety of other services built in. DLLs can also be mounted to extend this process's functionality. This process is provided in these versions:

`io-pkt-v4`

- Included services: UDP, TCP, IPv4, PPP, BPF, TUN, TAP
- Loadable modules: Qnet, Packet Filter (PF), autoIP, SLIP, shim

`io-pkt-v4-hc`

- Included services: UDP, TCP, IPv4, PPP, BPF, TUN, TAP, IPSec, HW crypto offload
- Loadable modules: Qnet, PF, autoIP, SLIP, shim

`io-pkt-v6-hc`

- Included services: IPv6, IPv4, UDP, TCP, PPP, BPF, TUN, TAP, IPSec, HW crypto offload
- Loadable modules: Qnet, PF, autoIP, SLIP, shim

As the `io-pkt` process executes in the application space, it's possible to launch more than one instance of it (see the `io-pkt` documentation). This ability is limited only by the management of the hardware interfaces, as typically only one driver instance (in one `io-pkt` instance) would manage one hardware interface.

The `io-pkt` manager has three defined driver interface APIs: native, shim, and BSD. Shim and BSD are typically used only to support legacy drivers and aren't intended for new driver development.

Native

This is the primary `io-pkt` API for new driver development. Native `io-pkt` drivers are noted by their file name prefix `devnp-`. This document will focus on this interface. Developing a driver for this interface provides the best integration with `io-pkt` (standard statistics and driver management) along with the best performance.

Shim

This is the interface supplied to support legacy io-net drivers (io-net was the network manager that io-pkt replaced). This interface allows io-net drivers to be loaded by io-pkt without modification. The additional DLL devnp-shim.so must be available, and is automatically loaded by io-pkt when an io-net driver is loaded. The io-net drivers are identified by their file name prefix devn-.

BSD

This is the interface supplied to support legacy NetBSD drivers. It doesn't support a native NetBSD binary, but does allow that driver's source to be recompiled for QNX Neutrino via a supplied BSD abstraction layer library which maps NetBSD library functions to QNX Neutrino functions where required. BSD drivers also use the file name prefix devnp-.

Drivers using any of these interfaces can be mounted and unloaded (unmounted or destroyed) dynamically. Loading and unloading can be performed in the following way:

- Loading drivers when starting io-pkt:

  ```
  io-pkt-v4-hc -d name or full path of driver binary [option,[option,..]] [-d
   ....]
  ```

  where *name* is the unique part of the driver file name (for example speedo vs devnp-speedo.so). Each unique driver instance (different option set or different driver) is specified with another -d driver option on the command line.

- Loading drivers after io-pkt process is running:

  ```
  mount -T io-pkt [-o option[,option,...]] full path of driver
  ```

  You don't have to specific the devnp-shim.so DLL on the command line of either io-pkt or mount; it's loaded automatically if needed.

- Unloading driver DLLs:

  ```
  ifconfig interface_name destroy
  ```

  Once all the interfaces managed by a driver have been destroyed, the driver DLL is unloaded, unless there are special actions taken by the driver to stay resident.

- Unloading shim driver DLLs: Along with ifconfig *interface_name* destroy, io-net shim drivers also support umount:

  ```
  umount /dev/io-net/interface_name
  ```

`io-pkt` architecture

As described in the *System Architecture* under Networking Architecture, Threading Model, `io-pkt` is a multithreaded process. We recommend that you read that section before continuing.

From this section we have at least the following thread types:

Stack context

This isn't really a specific thread, but a context of code that's single-threaded and can't be executing in multiple POSIX threads at the same time. It handles the main processing of `io-pkt`, such as the `io-pkt` resource manager dispatcher (see QNX Neutrino resource managers), which manages BSD socket API related operations, TCP/IP stack code and layer 3 code.

Since this context of code is single threaded, you must *never* block it. If you block this context of code, all the operations it performs (such as the resource manager) will be blocked until it's released. If during testing of your driver the `ifconfig` utility becomes blocked on `io-pkt` and doesn't terminate with the expected output, there is a good chance that you have blocked the stack context in your driver.

While single threaded, the stack context can manage blocking operations. This is via *pseudo-threading*. A "stack" is maintained per pseudo-thread. If a pseudo-thread is going to block, it's put to sleep, to be woken when the required condition is met. Only pseudo-threads within the stack context can yield execution to each other. You can't use sleep and wake routines outside of the stack context. This includes functions that call these routines. If you use these function outside the stack context, `io-pkt` can become unstable or fault. For more information on blocking and interacting with the stack context see the "Stack context" section below.

`io-pkt`-created threads

These are real POSIX threads created by `io-pkt`. In practice, `io-pkt`-created threads can be these types:

- Main thread (thread name `io-pkt main`, as listed in with the `pidin` utility's `threads` option.

  This is the thread created at `io-pkt` process startup to initialize `io-pkt`. It's generally idle after the `io-pkt` is initialized and its worker posix threads are started. It will never be the stack context. While generally idle, there is a way to leverage it for network driver blocking operations if needed (see blockop in the later sections).

- `io-pkt` worker thread (thread name `io-pkt#0x0`*N*)

These are threads created by `io-pkt` to service interrupts. As discussed in the "Threading Model" of the Network Architecture guide, one thread is created per CPU. You can use an `io-pkt` option to create more or less threads based on unusual conditions, but its optimal format is one POSIX thread per CPU.

The naming of the thread is the default `io-pkt` thread naming for any `io-pkt` managed thread, so this naming doesn't absolutely identify one of these threads (see "User-created `io-pkt` managed thread" below). The `io-pkt` worker threads will also execute the stack context code, and are the only threads that can execute the stack context code. Only one of these thread can execute this code at a time. The stack context may also migrate between the `io-pkt` worker threads, depending on the circumstances.

User-created threads include the following:

- `io-pkt` user-created `io-pkt` tracked thread (thread name is specified by the user; the default is `io-pkt#0x0`*N*")

  These are POSIX threads that were created by a dynamically loaded library (driver or other) or a thread created by an internal `io-pkt` service. These threads are created and managed by an `io-pkt`-specific POSIX thread API and can't execute the stack context code.

  An `io-pkt` internal service example is the PPP read thread (identified as such in the `pidin threads` output). These threads are typically created to handle blocking operations (such as a blocking *read()*) in the PPP case. This keeps the stack context from becoming blocked.

  User-created `io-pkt`-managed threads should always have a thread name assigned to them to make it easy to identify them during debugging situations. If they aren't named, they can be hard to distinguish from one another as well as from the `io-pkt` worker threads, as by default they use the same naming convention. While these threads can't perform operations that manipulate the pseudo-threads of the stack context, they can allocate and free mbufs and clusters and other memory objects via the `io-pkt` memory management. They can't however perform memory allocation as a `M_WAITOK` operation (they must always use `M_NOWAIT`). Using `M_WAITOK` would engage the pseudo-thread code in the stack context.

- User-created `io-pkt` thread "Not Tracked" (Default thread name is undefined)

  These are threads created by the user using the default `libc` Posix thread API. We don't recommended that you create threads in this manner as there will be no io-pkt context associated with them. This means that they can't allocate memory using io-pkt's memory management and can't be managed via io-pkt's thread synchronization mechanisms.

Typically the reason these threads may exist is during integration of third party code or library functions that create threads for specific tasks (for example USB insertion and removal event thread via libusbdi), or legacy io-net drivers that created receive threads.

If a thread is created using this API, it should operate in a manner that abstracts it from io-pkt API functions, so they aren't performed by this thread. For example, if a mbuf or cluster memory buffer needs to be created and managed, this thread could modify the data in the buffer, but couldn't allocate or free this buffer. Thread management must also be done by user code (starting/terminating and synchronizing), because io-pkt isn't aware of this thread. As with the io-pkt managed threads, these threads should be named.

For ease of debugging, it's recommended that any user-created io-pkt POSIX threads be named (see the *pthread_setname_np()* function). This will make it easy to identify the service that the thread provides in the pidin threads output if there are any problems.

Integration considerations

When coding a new io-pkt driver or porting existing driver code, you will want to consider how best to integrate it with io-pkt. The io-pkt program is optimized in such a manner that the preferred driver architecture doesn't require the creation of any driver-specific POSIX threads. This is to minimize thread switching in high-bandwidth situations (including forwarding between interfaces). Most io-pkt driver callback functions can potentially be called from the stack context. If this is the case, any time you spend in your driver is time that is potentially blocking other network operations from occurring.

The io-pkt manager is POSIX multi-threaded and can perform packet-forwarding operations in multiple threads simultaneously. The issue is that you can't predict when your function is being called with the stack context or not (although it's possible to determine if you have been at that time), so you still need to consider that it can and may often occur.

Whether you need to create a thread or take other special steps will probably depend on a few considerations:

- Does your driver code block on certain operations for undefined periods of time?

- Does the hardware your driver manages perform additional functions beyond network packet TX and RX?
- Does your hardware integration contain many more steps than one interrupt per RX packet or set of RX packets which are DMA into a descriptor ring?

If so, you may have to consider some of the advanced topics described later in this appendix. If not, then it's likely you should be able to integrate your driver as close to the optimized architecture as possible without using additional threads.

Examples of blocking include:

- Calling a function that requires a message pass to another manager.

  A typical scenario is a read operation to the USB stack (`io-usb`) by a USB network driver, or a read operation to a serial port or character-based interface (for example `io-pkt` PPP code). In these cases, we don't know when data to RX will arrive, so this could result in blocking indefinitely. You can send a message to another process, but you will expect an immediate response.
- Performing a function not related to packet RX and TX, but that is time consuming, or for whatever reason (HW or otherwise), TX and RX routines are extremely time consuming.
- The HW requires that firmware be uploaded before it can function.

An example of your hardware managing additional functions could be that the hardware services a multipurpose BUS. Ethernet frames may just be one type of data passed on this BUS, encapsulated within specific framing associated with this BUS although the primary data passed is network data.

You may want to create a resource manager within `io-pkt` to allow other types of data along with the TCP/IP traffic to be passed on the BUS. In this case, we are optimizing the TCP/IP traffic over other frame types. An architectural alternative could be to create a dedicated process to manage the BUS and require the `io-pkt` driver to perform message-passing to communicate with the BUS manager. This would be a more system-wide BUS sharing consideration.

An example of complicated hardware integration could be an interface with limited or no support of optimizations such as DMA and descriptor ring support. It may require multiple operations to obtain packet data where each suboperation requires its own interrupt, or multiple status requests are required. This can be time-consuming and complicated to integrate. A thread dedicated to managing HW RX and potentially TX may be needed.

Typical PCI network driver

See the *Writing Network Drivers for `io-pkt`* appendix and the accompanying sample driver, *`sam.c`*.

Managing the TX queue during resource conflict or link failure

One of the items often overlooked in `io-pkt` drivers is restarting transmission if some kind of resource conflict/exhaustion occurred or the link state is down. When `io-pkt` calls the driver *if_start()* callback function, it expects the TX queue to be drained. If it isn't, it will not call this callback function again unless there's a new packet added to the output queue. Also if the link state is down, the TX queue can fill up with packets to be sent. When the link state is restored, `io-pkt` will wait until the next packet transmission to call the *if_start()* callback, so that the packets in the send queue are transmitted.

Often managing this behavior is overlooked and can be misinterpreted at runtime as a lost or dropped frame, which was retransmitted simply because another packet will likely be sent shortly afterward to cause the *if_start()* callback to be executed again.

Differences between the interface Up/Down state and the Link up/down state

The first place to start is the difference between the interface up and down state vs the link up and down state. The interface up and down state is reflected in the interface flags (`IFF_UP` and `IFF_DOWN`), which can be viewed by `ifconfig`. The link up and down state is reflected in the media flags and can be viewed by `ifconfig` under the “media:” heading, and can also be viewed with `nicinfo` under the heading “Link is down|up.” These states are managed independently of each other, and one can be up while the other is down and vice versa.

The interface state is set via `ifconfig` and its default is down until set up when configured up explicitly, or when an IP address is assigned to the interface. It's considered an advisory state, as it reflects whether the user has set the interface up or down, regardless of the link's state. If the interface state is marked down, TX packets are dropped (memory is freed) without being queued, and the application can receive the error `ENETDOWN`. Likewise, RX packets are dropped by *ether_input()* (which is called by the driver on RX).

The link state is set by the driver itself based on the status of the physical link. If the link state is down, no RX packets will arrive, but on TX, the behavior is driver-specific. The MII code may update the status to `io-pkt` as displayed by the routing socket, `ifconfig`, and `nicinfo`, but otherwise `io-pkt` takes no specific action. On TX, (provided that the interface state is up) the packet will be added to the interface send queue (if it isn't already full), your driver's *if_start()* function will be called, and what occurs with respect to the send queue will be driver-specific.

Managing the TX queue

As we saw in the driver sample above, on TX the *if_start()* driver callback obtains packets to transmit from the *ifp->if_snd queue*. Packets are added to the send queue regardless of link state or other HW resource issues. One of the first things done in *if_start()* is to set the interface flag `IFF_OACTIVE`. This flag defines whether the driver is actively attempting to transmit data. This is a driver-level flag and isn't limited to the context of the *if_start()* callback function itself. If this flag is set, `io-pkt` will not attempt to call the *if_start()* callback again.

The significance of this is what occurs if there aren't enough resources to TX the packet, or if the link state is down. What should be done?

If nothing is done, the driver clears `IFF_OACTIVE` and *if_start()* returns, the packets remain on the send queue and *if_start()* will not be called again until there's another packet to be sent, at which point everything is evaluated again as before. If the link remains down, the send queue can fill, and applications could start getting `ENOBUF` errors. The driver may first exhaust the TX descriptors. It all depends on how the driver was coded. It can also be possible to get into this state when the link state is up simply because the HW couldn't transmit the packets quickly enough, exhausting the TX descriptors. We probably want the driver to continue transmission when the hardware or descriptor ring is ready, rather than wait until `io-pkt` has another packet to add to the send queue.

What needs to be decided is what to do if packets can't be transmitted: whether to leave the packets in the buffer, for how long, and how often should the driver attempt to send them. These parameters are specific to the driver implementation, but here is how they can be applied.

A timer can be enabled with a callback function to execute the *if_start()* callback. So for example, if the hardware isn't ready:

```
static void
sam_kick_tx (void *arg)
{
    sam_dev_t *sam = arg;
    NW_SIGLOCK(&sam->ecom.ec_if.if_snd_ex, sam->iopkt);
    sam_start(&sam->ecom.ec_if);
}

...

void
sam_start(struct ifnet *ifp)
{
    .....
    if (callout_pending(&sam->tx_callout))
        callout_stop(&sam->tx_callout);
    ifp->if_flags_tx |= IFF_OACTIVE;   /* Actively sending data on interface */
    .....
    if (detected_issue) {
        /* Resources aren't ready or something else is wrong */
        /* Set a callback to try again later */
        callout_msec(&sam->tx_callout, 2, sam_kick_tx, sam);
        /* Actual timeout value can be configurable or vary based on
           implementation */
        /* Leave IFF_OACTIVE set so the stack doesn't call us again */
        NW_SIGUNLOCK(&ifp->if_snd_ex, sam->iopkt);
        return;
    }
    ...

    /* Successful execution of sam_start() */
    ifp->if_flags_tx &= ~IFF_OACTIVE;
    NW_SIGUNLOCK(&ifp->if_snd_ex, sam->iopkt);
    return;
}
```

You can also make a similar call when the link is detected up in your MII code. In this case, you may perform some queries to determine if there is data to be sent; you may want to check both the transmit descriptor list and the interface send queue:

```
    ...
    sam->cfg.flags &= ~NIC_FLAG_LINK_DOWN;
    if_link_state_change(ifp, LINK_STATE_UP);
    if (data_in_tx_desc || !IFQ_IS_EMPTY(&ifp->if_snd)){
        /* There is some data to send */
        if (callout_pending(&sam->tx_callout))
            callout_stop(&sam->tx_callout); /* Timer not needed calling
                                               if_start() callback directly. */
        NW_SIGLOCK(&ifp->if_snd_ex, sam->iopkt);
        sam_start(ifp);
    }
    ...
```

If you set this timer, it should be stopped if an `ifconfig` *interface_name* down occurs, or otherwise the *if_stop()* driver callback function is executed. When this occurs, the following can be called early in *if_stop()*:

```
static void
sam_stop(struct ifnet *ifp, int disable) {
    ...
    /* Lock out the transmit side */
    NW_SIGLOCK(&ifp->if_snd_ex, sta2x11->iopkt);
    if (callout_pending(&sam->tx_callout)) {
        callout_stop(&sam->tx_callout);
        /* We aren't in if_start() as it stops the callout */
        ifp->if_flags_tx &= ~IFF_OACTIVE;
    }
    for (i = 0; i < 10; i++) {
        if ((ifp->if_flags_tx & IFF_OACTIVE) == 0)
            break;
        NW_SIGUNLOCK(&ifp->if_snd_ex, sam->iopkt);
        delay(50);
        NW_SIGLOCK(&ifp->if_snd_ex, sam->iopkt);
    }
    if (i < 10) {
        ifp->if_flags_tx &= ~IFF_RUNNING;
        NW_SIGUNLOCK(&ifp->if_snd_ex, sam->iopkt);
    } else {
        /* Heavy load or bad luck. Try the big gun. */
        quiesce_all();
        ifp->if_flags_tx &= ~IFF_RUNNING;
        unquiesce_all();
    }
    ...
    /* Mark the interface as down and cancel the watchdog timer. */
    ifp->if_flags &= ~(IFF_RUNNING | IFF_OACTIVE);
    ifp->if_timer = 0;
    return;
}
```

The last point is stale data. These are packets that have accumulated in the send queue but can't be sent. How long should attempts to retransmit this data be made and when should the queue be flushed? You probably want to consider flushing the queue, as you probably don't want to send packets that have sat in the send queue for extended periods of time, as the data is probably out of date.

Above we've seen how to use a timer to resume transmission, or to use link state to resume transmission. This is based on the idea that the issues related to TX are sporadic and for short periods of time. A decision may have to be made when to declare the data stale as well as to stop data from being queued. We can flush the send queue, but we also want io-pkt to stop queuing packets, or the send queue will just fill up again.

Based on some kind of timing, if TX hasn't resumed, you can decide to purge the send queue. This can be managed by a higher level or at the driver level. If managed at the higher level, marking the interface down by clearing the IFF_UP interface flag will cause the send queue to be purged. At the driver level you can perform the same operation via:

```
IFQ_PURGE(&ifp->if_snd);
```

If the interface remains down, no new packets will be added to the send queue. If the interface is marked up, io-pkt will continue to add packets to the send queue. If the interface remains up, periodic purging may be needed if TX hasn't resumed at the hardware level.

Advanced driver integration topics

Blocking Operations

The io-pkt manager is optimized to minimize thread switching, and as mentioned in the architecture discussion previously, driver API callback functions can be called from the stack context. As the stack context is single-threaded, we can't have blocking operations being performed within the stack context. If a blocking operation occurs, you will block the stack context (io-pkt resource manager, protocol processing) for the duration of the time spent blocked.

If during testing of your driver, io-pkt or your driver seems unresponsive, try executing ifconfig. If it doesn't terminate with output and is blocked on the io-pkt process, there's a good chance that the stack context is blocked. Check the state of the threads in the pidin output. If any threads named 0x00, 0x01, and so on (depending on the number of CPU on your system) are blocked on a mutex, semaphore, condvar, or other resource manager, it means the stack context is blocked, and there may be some kind of deadlock or general blocking issue in the driver API callback functions.

What defines blocking? Basically any time spent in the driver API callback functions may potentially be time that io-pkt can't service the resource manager (applications), timers, and processing associated with the supported protocols in io-pkt. Time spent in the driver API callback functions should be as little as possible.

Some examples to consider are:

- Message passing with another resource manager.

  Many QNX Neutrino function calls result in a message's being sent to another resource manager. If the message being sent will not get a immediate response (a blocking read() or write() for example), you can block io-pkt. The typical example is a read operation; if it's a blocking read, the function call may not return until there is data to read.

- Locking resources.

  If the resource is already locked, can it potentially be locked for a long period of time, blocking the callback function while the driver waits to acquire the lock?

  Does your hardware require a service that can take a long period of time, such as loading the firmware?

Block Op

If the duration of the blocking scenario is known and within a few seconds or less, you can use the *blockop services*. Essentially this offloads an operation that may take some time to the `io-pkt` main thread (which is typically idle). Note that blockop is a shared service, and may have multiple operations scheduled. This is meant for occasional time-consuming operations (such as a firmware upload that occurs once), but not indefinite or long-term operations and not repetitive operations. It's a convenience service that handles the complicated management of the stack context pseudo-thread handling. As it performs these kinds of operations, it must be called from within the stack context. The callback function however isn't called from the stack context and shouldn't perform any operations that require the stack context or buffer management.

The example below is taken from the PPP data link shutdown processing. In this case, the *close()* function for the serial port resource manager takes an unusually long, but predictable, amount of time to reply the the message blocking the *close()* function. Since this is called in the stack context, it blocks other `io-pkt` operations until the *close()* returns. This code moves the execution of the *close()* into the main `io-pkt` thread, and pseudo-thread switches to other operations until the callback function returns. The *qnxppp_ttydetach()* function pseudo-thread switches at the *blockop_dispatch()* and resumes from the same point once the *qnxppp_tty_close_blockop()* function returns.

```
#include <blockop.h>

struct ppp_close_blockop {
    int qnxsc_pppfdrd;
    int qnxsc_pppfdrd2;
    int qnxsc_pppfdwr;
}

void qnxppp_tty_close_blockop(void *arg);
....

void qnxppp_tty_close_blockop(void *arg)
{
    struct ppp_close_blockop *pcb = arg;

    if(pcb->qnxsc_pppfdrd != -1)
        close(pcb->qnxsc_pppfdrd);
    if(pcb->qnxsc_pppfdrd2 != -1)
        close(pcb->qnxsc_pppfdrd2);
    if(pcb->qnxsc_pppfdwr != -1)
        close(pcb->qnxsc_pppfdwr);
}
```

```
int qnxppp_ttydetach(...)
{
    struct ppp_close_blockop pcb;
    struct bop_dispatch bop;
    .....

    pcb.qnxsc_pppfdrd = sc->qnxsc_pppfdrd;
    pcb.qnxsc_pppfdrd2 = sc->qnxsc_pppfdrd2;
    pcb.qnxsc_pppfdwr = sc->qnxsc_pppfdwr;

    bop.bop_func = qnxppp_tty_close_blockop;
    bop.bop_arg = &pcb;
    bop.bop_prio = curproc->p_ctxt.info.priority;
    blockop_dispatch(&bop);
    ....

    return;
}
```

Thread Creation

As stated above, there are several types of threads that can exist in an instance of `io-pkt`. The two types of threads created by driver or module developers from above are user-created threads that are either tracked (*nw_pthread_create()*) or not tracked (*pthread_create()*) by `io-pkt`. Regardless of how they're created, all POSIX threads created in `io-pkt` should be named for easier debugging.

Untracked threads

The only time you should be dealing with untracked threads is if you're using a library that creates threads for the services it provides. An example of this is the USB stack library (`libusbdi`), which can create a thread to call user-provided callback functions to handle device insertion and removal.

If your code creates a thread directly, you should create a tracked thread as described below. If you're calling library functions that create threads on your behalf, you must manage these threads in your module code, because `io-pkt` isn't aware of their existence. As stated under the `io-pkt` Architecture section, threads that aren't tracked can't allocate or free an mbuf or cluster, and can't call functions that perform any manipulation of the stack context pseudo-threading.

Tracked threads

If you're creating a thread in your `io-pkt` module, you should always use *nw_pthread_create()* rather than *pthread_create()*. The *nw_pthread_create()* function creates a thread that's tracked by `io-pkt`. This allows the thread to allocate and free mbuf and cluster memory buffers, and also provides a synchronization mechanism, this being the quiesce functionality, which either blocks all `io-pkt`-tracked POSIX threads for exclusive operations, or causes these threads to exit on shutdown.

All tracked POSIX threads must register a quiesce callback function (defined below). If your thread doesn't register a quiesce callback function, `io-pkt` can end up in a deadlock situation.

In the sample below, *nw_pthread_create()* is the same as *pthread_create()*, but for some considerations in the initialization function. The first consideration is naming the thread for easier debugging. The other is setting up the mechanism for your threads' quiesce handling where `io-pkt` requires all threads to block for an exclusive operation. This is required of all threads created with *nw_pthread_create()*.

Threads can be terminated via quiesce_block handling, or using the function *nw_pthread_reap(tid)*, where *tid* is the thread ID of your tracked thread. The *nw_pthread_reap()* can't be called by the thread specified by the *tid* argument (i.e., a thread can't reap itself).

Both *nw_pthread_create()* and *nw_pthread_reap()* must be called from the stack context.

Below is an example where the user-created tracked thread creates a resource manager. The structure of your driver can be different, but the main point is that your quiesce callback function must cause your tracked thread to call *quiesce_block()*.

```
#include <nw_thread.h>

static sam_thread_init(void *arg)
{
    struct nw_work_thread *wtp;
    sam_dev_t *sam = (sam_dev_t *)arg;

    pthread_setname_np(gettid(), "sam workthread");
    wtp = WTP;

    ...

    if ((sam->code = pulse_attach(sam->dpp, MSG_FLAG_ALLOC_PULSE, 0,
         sam_pulse_func, NULL)) == -1)
    {
        log(LOG_ERR, "sam: pulse_attach(): %s", strerror(errno));
        return errno;
    }
    if ((sam->coid = message_connect(sam->dpp, MSG_FLAG_SIDE_CHANNEL)) == -1)
    {
        pulse_detach(sam->dpp, sam->code, 0);
        log(LOG_ERR, "sam: message_connect(): %s", strerror(errno));
        return errno;
    }
    wtp->quiesce_callout = sam_thread_quiesce;
    wtp->quiesce_arg = sam;

    ...

   return EOK;
}

static int sam_pulse_func(message_context_t *ctp, int code, unsigned flags,
                          void *handle)
{
    /* If the die argument is 1, the user thread will terminate in queisce_block */

    quiesce_block(ctp->msg->pulse.value.sigval_int);
    return 0;
}

static void sam_thread_quiesce(void *arg, int die)
{
sam_dev_t *sam = (sam_dev_t *)arg;

        MsgSendPulse(sam->coid, SIGEV_PULSE_PRIO_INHERIT, sam->code, die);
}

static void *sam_thread(void *arg)
{
```

```
sam_dev_t *sam = (sam_dev_t *)arg;
        dispatch_context_t *ctp;

        if ((ctp = dispatch_context_alloc(sam->dpp)) == NULL {
                ...
                return NULL;
        }
        while(1) {
                if ((ctp = dispatch_block(ctp)) == NULL) {
                        ...
                        break;
                }
                dispatch_handler(ctp);
        }
        return NULL;
}

...

/* Likely in the sam_attach() interface attach driver callback function */

/*Need a thread to handle blocking or other special circumstance scenario */

    if (nw_pthread_create(&sam->worker_tid, NULL, sam_thread, sam, 0,
        sam_thread_init, sam) != EOK)
    {
        log(LOG_ERR, "sam: nw_pthread_create() failed\n");
        /* Clean up and likely return -1 */
    }

/* Likely in the sam_detach() interface detach driver callback function */

    ...
    if (nw_pthread_reap(sam->worker_tid))
        log(LOG_ERR, "%s(): nw_pthread_reap() failed\n", __FUNCTION__);
    ...
```

Quiesce handling

Quiesce handling is required by all threads that are created by *nw_pthread_create()*. The purpose of this functionality is to allow `io-pkt` to quiesce (quiet) all threads for an exclusive operation. It also provides a mechanism for terminating the thread.

The basic structure of the mechanism is the quiesce callback function provided by the driver (example above), and the *quiesce_block()* `io-pkt` function that the tracked thread is required to call. The quiesce callback function is executed by `io-pkt` (otherwise called from the stack context via the *quiesce_all()* function). This callback function provides some kind of mechanism to trigger the tracked thread to call the function *quiesce_block()* with the *die* argument provided to the callback function. This argument determines if the thread blocks (*die* = 0) or terminates (*die* = 1).

If the *quiesce_block()* function isn't called by the tracked thread, `io-pkt` (and thus the stack context) will be blocked in *quiesce_all()* until it does, as *quiesce_all()* is intended to block all worker threads until *unquiesce_all()* is called to resume the tracked threads. The *unquiesce_all()* function must also be called from the stack context.

If the *die* argument is 1, your thread will terminate in *quiesce_block()*. Before you call *quiesce_block()*, you may need to free any dependencies associated with that thread.

As well, if *die* is 0, your thread will block for a short period of time. You may have HW integration issues to consider that could be affected by this blocking.

You may want to have some code around the *quiesce_block()* to handle this, such as disable and enable interrupts or other hardware considerations. These considerations would be implementation-specific.

If we continue from the example above, the callback function provided will send a pulse to a channel managed by the tracked thread (its resource manager). That pulse will trigger another callback function that's executed by the tracked thread. This function calls *quiesce_block()* with the *die* argument provided.

Don't call *queisce_block(die)* to stop a thread without its being triggered by your quiesce callback function; if you want to terminate your tracked thread, call *nw_pthread_reap()* from the stack context.

Periodic timers

Network drivers frequently need periodic timers to perform such housekeeping functions as maintaining links and harvesting transmit descriptors. The preferred way to set up a periodic timer is via the callback API provided by `io-pkt`. This API is used to call a user-defined function after a specified period of time. You can call *callout_** functions in `io-pkt` driver API callbacks, or *nw_pthread_create()*-created `io-pkt` threads. The callout function will be called from the stack context.

The callout data type is `struct callout`, and includes the following functions:

`void callout_init (struct callout *c)`

Initialize the callout structure.

`void callout_msec(struct callout *c, int msec, void (*func)(void *), void *arg)`

Schedule a function to be called after the specified number of msec have expired.

`void callout_stop(struct callout *c)`

Cancel the callout.

`callout_pending(struct callout *c)`

If true, a callout is pending; if false, a callout isn't pending.

Here's an example:

```
struct sam_dev {
    ...
    /* Declare a type callout in your driver device structure */
    /* Unique to this interface */
    struct callout my_callout;
    ...
};

static void
my_function (void *arg)
```

```
{
     struct sam_dev *sam = arg;

     /* Do something if the timer expires */

     /* We may want to arm the callout again if we want my_function() to be
        called on a regular interval. */
     callout_msec(&sam->my_callout, 5, my_function, sam);
{


/* Before it's used, it must be initialized */
/* This can be in the if_init() or if_attach() callback for example */

callout_init(&sam->my_callout);   /* Initialize callout */

/* Once initialized it can be used */

/* Call my_function() in 5 ms */
callout_msec(&sam->my_callout, 5, my_function, sam);

callout_stop(&sam->my_callout);  /* Cancel the callout */

if (callout_pending(&sam->my_callout)) {   /* Is the callout armed */
    /* action if pending */
} else {
    /* action if not pending */
}
```

Driver doesn't use an RX interrupt

When the driver isn't notified via an interrupt that a packet has arrived, you will need to mimic this functionality in your driver. There are different approaches to this, with different limitations. In your *nw_pthread_create()* thread, you can either call *if_input()* directly, or simulate the ISR.

Calling *if_input()* directly has limitations, as your interface will not be able to support fastforward feature or bridging between interfaces if you're considering these features for the future. You would prepare the mbuf in the same manner as the sample's "process interrupt" function, and end with calling *if_input()*. The *if_input()* function executed in your (nw_pthread)thread will cause the packet to be queued, and an event will trigger the main `io-pkt` threads to process the packet.

The other method allows fastforward and bridging to work as in other `io-pkt` drivers. In this case, you will enqueue your packets in your nw_pthread, and trigger the event directly in your code to cause your "process interrupt" `io-pkt` callback to execute in the same way it would if an ISR had occurred. In the "process interrupt" callback, you would dequeue the packet from your internal driver queue, prepare the mbuf in the same manner as the sample, and execute *if_input()*. In this case, *if_input()* is executed in the `io-pkt` callback rather than in your nw_pthread.

For this, you will define a process interrupt callback, along with an enable-interrupt callback as you would with an ISR. The difference is how the *interrupt_queue()* is applied. In your case, you will have a queue that's accessed by two different threads, the one receiving the packet from the HW, and the other "process interrupt" passing the packet to upper layers in `io-pkt`. You will want a mutex protecting this queue so it's modified by only one thread at a time. You will also want to protect the event notification mechanism *interrupt_queue()* is using.

In your driver thread, you will lock the mutex, check if the queue is full, and if not, enqueue the packet in your internal queue. You will now call *interrupt_queue()* in your thread. If *evp* (event structure) isn't `NULL`, you will send this event yourself in your thread:

```
MsgSendPulse(evp->sigev_coid, evp->sigev_priority, evp->segev_code,
             (int)evp->sigev_value.sival_ptr);
```

Once you have done this, you would unlock your mutex. The remainder of your function will be hw management or descriptor management.

The `io-pkt` manager will now schedule your "process interrupt" callback to execute. In your process interrupt callback, you will loop dequeueing packets until the queue is empty. First you will lock your mutex for your internal queue, and attempt to dequeue a packet. If you did dequeue a packet, unlock your mutex, and call *if_input()* (provided the mbuf is prepared as required) and go back to the top of the loop. If there is no packet to dequeue (*IF_DEQUEUE()* returns `NULL`), break out of your loop and return *without unlocking your mutex*. You don't want to unlock your mutex here because we don't want your receive thread to call *interrupt_queue()* at this point. If it did, it would return `NULL` because you're currently processing a packet. You will unlock your internal mutex in the "enable interrupt" callback. This way a new `evp` structure will be returned when your receive thread can continue as you are finished processing packets.

Your mbuf can be prepared either in your receive thread or the "process interrupt" callback. It just depends on whether you want to store the fully formed mbuf in your internal queue or partial buffers to be formatted later.

In the receive thread:

```
struct sigevent *evp;

pthread_mutex_lock(&driv->rx_mutex);
if (IF_QFULL(&driv->rx_queue))
{
    m_freem(m);
    ifp->if_ierrors++;
    ...->stats.rx_failed_allocs++;
}
else
{
   IF_ENQUEUE(&driv->rx_queue, m);
}

if (!driv->rx_running)
{
   //RX_running is mimicking interrupt masking.
   // This is for future compatibility when using interrupt_queue()

   driv->rx_running = 1;
   evp = interrupt_queue(driv->iopkt, &driv->inter);
   if (evp != NULL)
   {
       MsgSendPulse( evp->sigev_coid, evp->sigev_priority, evp->sigev_code,
                    (int)evp->sigev_value.sival_ptr);
    }
}

pthread_mutex_unlock(&driv->rx_mutex);
```

In the main code:

```
int your_process_interrupt( void *arg, struct nw_work_thread *wtp)
{
    driver_dev_t   *driv = arg;
    struct ifnet  *ifp;
    struct mbuf *m;

    ifp = &driv->ecom.ec_if;
    while (1)
    {
        pthread_mutex_lock(&driv->rx_mutex);
        IF_DEQUEUE(&driv->rx_queue, m);
        if (m!= NULL)
        {
             pthread_mutex_unlock(&driv->rx_mutex);
             ...
             Prepare mbuf if needed
             ...
             (*ifp->if_input)(ifp, m);
        }
        else
        {
             /* Leave mutex locked to prevent any enqueues; unlock in enable */
             break;
        }
    }
    return 1;
}

int your_enable_interrupt (void *arg)
{
    driver_dev_t *driv = arg;
    ...
    driv->rx_running = 0;
    pthread_mutex_unlock(&driv->rx_mutex);
    return 1;
}
```

Glossary

AES

An abbreviation for *Advanced Encryption Standard)*.

BPF

An abbreviation for *Berkley Packet Filter*.

BSS

An abbreviation for *Basic Service Set*. Also known as *Infrastructure Mode*.

CA

An abbreviation for *Certification Authority*.

EAP-TLS

An abbreviation for *Extensible Authentication Protocol - Transport Layer Security*.

IBSS

An abbreviation for *Independent Basic Service Set*.

`mbuf`

An abbreviation for *memory buffer*, the internal representation of a packet used by NetBSD and `io-pkt`.

NAT

An abbreviation for *Network Address Translation*.

`npkt`

The name of the internal representation of a packet used by `io-net`.

SA

An abbreviation for *Security Association*.

SOHO

An abbreviation for *Small Office, Home Office*.

SPD

An abbreviation for *security policy database*.

spoofing

The faking of IP addresses, typically for malicious purposes.

SSID

An abbreviation for *Service Set Identifier*.

STP

An abbreviation for *Spanning Tree Protocol*.

TDP

An abbreviation for *Transparent Distributed Processing*, the QNX Neutrino native networking (Qnet) that lets you access resources on other QNX Neutrino systems as if they were on your own machine.

TKIP

An abbreviation for *Temporal Key Integrity Protocol*.

TLS

An abbreviation for *Transport Layer Security*.

TTLS

An abbreviation for *Tunneled Transport Layer Security*.

UDP

An abbreviation for *User Datagram Protocol*.

WAP

An abbreviation for *Wireless Access Point*. Also known as a *base station*.

WEP

An abbreviation for *Wired Equivalent Privacy*.

WLAN

An abbreviation for *Wireless Local Area Network*.

WPA

An abbreviation for *Wi-Fi Protected Access*.

Index

J

K

L

M

N

O

P

Q

R

S

T

U

W