High Availability Framework Developer'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: Thursday, May 22, 2014

Table of Contents

About This Guide **7**
- Typographical conventions 8
- Technical support 10

Chapter 1: Introduction **11**
- Where's the problem? 12
- A software foundation for HA 13

Chapter 2: What's in the High Availability Framework? **15**

Chapter 3: The QNX Neutrino Approach to HA **17**
- The reset “solution” 18
  - Traditional RTOS architecture 18
- Modularity means granularity 20
  - Intrinsic HA 20

Chapter 4: Using the High Availability Manager **23**
- HAM hierarchy 24
  - Entities 24
  - Conditions 25
  - Actions 27
  - Action Fail actions 28
  - Multistaged recovery 30
- State of the HAM 31
  - Example of the view shown in /proc/ham 31
- HAM API 34
  - Connect/disconnect functions 34
  - Attach/detach functions 35
  - Entity functions 38
  - Condition functions 39
  - Action functions 41
  - Action fail functions 44
  - Functions to operate on handles 46
- A client example 47
- Starting and stopping a HAM 50
  - Stopping a HAM 50
  - Control functions 51
  - Verbosity control 52
  - Publishing autonomously detected conditions 52

Publish state transitions 52
Publish other conditions 53
Subscribing to autonomously published conditions 53
Trigger based on state transitions 53
Trigger based on specific published condition 54

Chapter 5: Using the Client Recovery Library 55
MsgSend*() functions 56
Other covers and convenience functions 57
HA-awareness functions 57
I/O covers 57
Convenience functions 58
A simple example 60
State-reconstruction example 63

Chapter 6: HAM API Reference 65
ham_action_control() 69
ham_action_execute() 72
ham_action_fail_execute() 75
ham_action_fail_log() 78
ham_action_fail_notify_pulse(), ham_action_fail_notify_pulse_node() 80
ham_action_fail_notify_signal(), ham_action_fail_notify_signal_node() 83
ham_action_fail_waitfor() 86
ham_action_handle(), ham_action_handle_node() 88
ham_action_handle_free() 90
ham_action_heartbeat_healthy() 92
ham_action_log() 94
ham_action_notify_pulse(), ham_action_notify_pulse_node() 96
ham_action_notify_signal(), ham_action_notify_signal_node() 99
ham_action_remove() 102
ham_action_restart() 104
ham_action_waitfor() 107
ham_attach(), ham_attach_node() 110
ham_attach_self() 114
ham_condition() 117
ham_entity_control() 121
ham_condition_handle(), ham_condition_handle_node() 123
ham_condition_handle_free() 125
ham_condition_raise() 127
ham_condition_remove() 129
ham_condition_state() 131
ham_connect(), ham_connect_nd(), ham_connect_node() 134
ham_detach() 136
ham_detach_name(), ham_detach_name_node() 138

ham_detach_self() 140
ham_disconnect(), ham_disconnect_nd(), ham_disconnect_node() 142
ham_entity(), ham_entity_node() 144
ham_entity_condition_raise() 146
ham_entity_condition_state() 148
ham_entity_control() 150
ham_entity_handle(), ham_entity_handle_node() 153
ham_entity_handle_free() 155
ham_heartbeat() 157
ham_stop(), ham_stop_nd(), ham_stop_node() 159
ham_verbose() 161

Chapter 7: Client Recovery Library Reference 163
ha_attach() 164
ha_close() 167
ha_connection_ctrl() 168
ha_ConnectAttach(), ha_ConnectAttach_r() 170
ha_ConnectDetach(), ha_ConnectDetach_r() 172
ha_creat(), ha_creat64() 174
ha_detach() 176
ha_dup() 178
ha_fclose() 180
ha_fopen() 181
ha_open(), ha_open64() 183
ha_ReConnectAttach() 185
ha_reopen() 187

Chapter 8: Examples 189
Simple restart 190
Compound restart 192
Death/condition notification 194
Heartbeating clients (liveness detection) 197
The process is made to heartbeat 200
Process starvation 203
Thread 2 is made to heartbeat 205
Glossary 209

About This Guide

The High Availability Framework *Developer's Guide* describes how to build robust high-availability software running on the QNX Neutrino RTOS.

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

If you want to:	Go to:
Find the introduction	*Introduction* (p. 11)
Learn about the main components	*What's in the High Availability Framework?* (p. 15)
Understand the benefits of a software-oriented approach to High Availability	*The QNX Neutrino Approach to HA* (p. 17)
Get an overview of the HAM and Guardian “watchdogs”	*Using the High Availability Manager* (p. 23)
Find out which standard QNX Neutrino library functions have HA covers	*Using the Client Recovery Library* (p. 55)
Look up a HAM API function (e.g., *ham_attach()*)	*HAM API Reference* (p. 65)
Look up a convenience function (e.g., *ha_recover()*)	*Client Recovery Library Reference* (p. 163)
See sample code listings for handling various HA scenarios	The *Examples* (p. 189) appendix
Look up a special term used in this guide	*Glossary*

For an overview of the QNX Neutrino RTOS, see the *System Architecture* guide.

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 typically use a forward slash (/) as a delimiter in 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
Introduction

The ideal High Availability (HA) system is one that remains up and running *continuously*, uninterrupted for an indefinite period of time. In practical terms, HA systems strive for “five nines” availability, a metric referring to the percentage of uptime a system can sustain in a year; 99.999% uptime amounts to about five minutes downtime per year.

Where's the problem?

Obviously, systems fail. For one reason or another, systems aren't as available for use as their users and designers would like them to be. Of all the possible causes of system failure — power outages, component breakdowns, operator errors, software faults, etc. — the lion's share belongs to software faults.

Many HA systems try to address the problem of system failure by turning to *hardware* solutions such as:

- rugged hardware
- redundant systems/components
- hot-swap CompactPCI components
- clustering

But if so many system crashes are caused by *software* faults, then throwing more hardware at the problem may not solve it at all. What if the system's memory state isn't properly restored after recovery? What if yours is an HA system (e.g., a consumer appliance) where redundant hardware simply isn't an option? Or what if your particular HA system is based on a custom chassis for which a PCI-based HA “solution” would be pointless?

A software foundation for HA

Most system designers wouldn't think of using a “standard” desktop PC as the foundation for an effective HA system. Apart from the reliability issues arising from the hardware itself, the *underlying software* isn't meant for continuous operation. When desktop operating systems and applications need to be patched or upgraded, most users expect to reboot their machines. Unfortunately, they might also have become accustomed to rebooting as part of their daily operations!

But in an HA system, various software components may need to be upgraded *on a live system*. Individual modules should be readily accessible for analysis and repair, without jeopardizing the availability of the system itself.

In our view, effective HA systems must address the main problem — software faults — through a modular approach to system design and implementation. Based on a microkernel architecture, the QNX Neutrino RTOS not only helps isolate problem areas throughout the system, but also ensures complete independence of system components. Each component enjoys full MMU-based memory protection. And system-level modules such as device drivers benefit from the same isolation and protection as any other process. You can start and stop a driver, networking protocol, filesystem, etc., without touching the kernel. A microkernel RTOS inherently keeps the *single point of failure* (SPOF) number as low as possible.

QNX Neutrino High Availability Framework provides a reliable software infrastructure on which to build highly effective HA systems. In addition to support for hardware-oriented HA solutions (e.g., CompactPCI as well as custom hardware), you also have the tools to isolate and even repair software faults before they occur throughout your entire system.

For example, suppose a device driver crashes because it tried to write to memory that was allocated to another process. The MMU will alert the microkernel, which in turn will alert the High Availability Manager (HAM). A HAM can then restart the driver. In addition, a dump file can be generated for postmortem analysis.

Viewing this dump file, you can immediately determine which line of code is the culprit and then prepare a fix that you can download to all other units in the field before they run into the same bug. With a conventional OS, a rogue driver may run for days before the system becomes corrupted enough to fail — and then it's too late to identify the problem, let alone dynamically install an upgraded driver!

A HAM can perform a multistage recovery, executing several actions in a certain order. This technique is useful whenever strict dependencies exist between various actions in a sequence, so that the system can restore itself to the state it was in before a failure.

Equipped with the QNX Neutrino RTOS itself, as well as the special tools and API in the High Availability Framework, you should be able to anticipate the kinds of problems that are likely to happen, isolate them, and then plan accordingly. In other words, assuming that failure will occur, you can now design for it and build systems that can recover intelligently.

Chapter 2
What's in the High Availability Framework?

The QNX Neutrino High Availability Framework consists of the following main components:

QNX Neutrino RTOS

We're not just trying to be thorough by listing the OS itself here! And it's first in the list for good reason — the QNX Neutrino microkernel architecture inherently provides a robust environment for building highly reliable applications. Many of the particular features required in an HA application — system stability, isolation of software modules, dynamic upgrading of software components, etc. — are already included in the OS.

The microkernel provides system-wide stability by offering full memory protection to all processes. And there's very little code running in kernel mode that could cause the microkernel itself to fail. All individual processes, whether applications or OS services — including device drivers — can be started and stopped dynamically, without jeopardizing system uptime.

For more on the suitability of the QNX Neutrino RTOS for HA, see the *next chapter* (p. 17) in this guide.

High Availability Manager (HAM)

A HAM is a “smart watchdog” — a highly resilient manager process that can monitor your system and perform multistage recovery whenever system services or processes fail or no longer respond.

As a self-monitoring manager, a HAM is resilient to internal failures. If, for whatever reason, the HAM itself is stopped abnormally, it can immediately and completely *reconstruct its own state* by handing over to a mirror process called the Guardian.

For details on the HAM, see the chapter *Using the High Availability Manager* (p. 23) in this guide.

HAM API

The HAM API library of more than 35 *ham_*()* functions gives you a simple mechanism to talk to a HAM. This API is implemented as a thread-safe library you can link against.

You use the API to interact with a HAM in order to begin monitoring processes and to set up the various conditions (e.g., the death of a server) that will trigger certain recovery actions.

For descriptions of the functions in the HAM API, see the *HAM API Reference* (p. 65) chapter in this guide.

Client Recovery Library

The client recovery library provides a drop-in enhancement solution for many standard `libc` I/O operations. The HA library's cover functions provide automatic recovery mechanisms for failed connections that can be recovered from in an HA scenario.

For descriptions of the client library functions, see the *Client Recovery Library Reference* (p. 163) chapter in this guide.

Examples

You'll find several sample code listings (and source) that illustrate such tasks as restarting, heartbeating, and more. Since the examples deal with some typical fault-recovery scenarios, you may be able to easily tailor this source for your HA applications.

For details, see the *Examples* (p. 189) appendix in this guide.

Chapter 3
The QNX Neutrino Approach to HA

Let's compare how QNX Neutrino and other OSs approach high availability.

The reset “solution”

Traditional approaches to dealing with software malfunctions have included such mechanisms as:

Hardware/software watchdog

This is a piece of hardware that's known to be fault-free. It triggers code to check the sanity of the system. This sanity check usually involves examining a set of registers that are continuously updated by properly functioning software components. But when one of the components isn't working properly, the system is reset.

Manual operator intervention

Many systems aren't designed to include an automatic fault detection, but rely instead on a manual approach — an operator who monitors the health of the system. If the system state is deemed invalid, then the operator takes the appropriate action, which usually includes a system reset.

Memory constraint faulting

Several operating systems (and hardware platforms) include features that let you generate a fault when a program accesses memory that isn't yours. Once this occurs, the program becomes unreliable. With most realtime executives, the result is that the system must be reset in order to return to a sane operating state.

All of these approaches are relatively successful at detecting a software fault. But the net result of this detection, especially when faced with a multitude of faults in several potentially separate software components, is the rather drastic action of a system reset.

Traditional RTOS architecture

One of the principal reasons for this lack of graceful recovery is the monolithic architecture of a traditional realtime embedded system. At the heart of most of these systems lies a *realtime executive* — a single memory image consisting of the RTOS itself and often numerous tasks.

Since all tasks — including critical system-level services — share the very same address space, when the integrity of one task is called into question, the integrity of the entire system is at risk. If a single component such as a device driver fails, the RTOS itself could fail. In HA terms, each software component becomes a single point of failure (SPOF).

The only sure recovery mechanism in such an environment is to reset the system and start from scratch.

Such realtime systems present a very low granularity of fault recovery, making the HA procedure of planning for and dealing with failure seemingly straightforward (a system reset), yet often very costly (in terms of downtime, system restoration, etc.). For some embedded applications, a reset may involve a specialized, time-consuming procedure in order to restore the system to full operation in the field.

Modularity means granularity

What is really needed here is a more modular approach. System architects often de-couple and modularize their systems from a design/implementation point of view. Ideally, these modules would be the focus not only of the design, but also of the fault-recovery process, so that if one module malfunctions, then only that module would require a reset — the integrity of the rest of the system would remain intact. In other words, that particular module wouldn't be a SPOF.

This modular approach would also help us address the fact that the mean time to repair (MTTR) for a system reboot is a magnitude larger than the MTTR for replacing a single running task.

This type of increased granularity on the recovery of individual tasks is precisely what the QNX Neutrino microkernel offers. The architecture of the QNX Neutrino RTOS itself provides so many intrinsic HA features that many QNX Neutrino users take them for granted and often design recoverability into their systems without giving it a second thought.

Let's look briefly at the key features of QNX Neutrino and see how system designers can easily make use of these builtin HA-ready features to build effective HA systems.

Intrinsic HA

Three key factors of the QNX Neutrino architecture contribute directly to intrinsic HA:

Microkernel

Only a few essential services are provided (e.g., message passing and realtime scheduling). The result is a robust, dependable system — fewer lines of code in the kernel reduce the probability of OS errors.

Also, the kernel's fixed-priority preemptive scheduler ensures a *predictable* system — there are fewer HA software paths to analyze and deal with separately.

POSIX process model

This means full MMU-supported memory protection between system processes, making it easy to isolate and protect individual tasks.

The process model also offers *dynamic* process creation and destruction, which is especially important for HA systems, because you can more readily perform fault detection, recovery, and live upgrades in the field.

The POSIX API provides a standard programming environment and can help achieve system simplification, validation, and verification.

In addition, the process model lets you easily monitor external tasks, which not only aids in fault detection and diagnosis, but also in service distribution.

Message passing

In the QNX Neutrino RTOS, all interprocess communication happens through standard message passing. For HA systems, this facilitates task decoupling, task simplification, and service distribution.

Local and network-remote messaging is *identical* and practically transparent for the application. In a network-distributed HA system, the QNX Neutrino message-based approach fosters replication, redundancy, and system simplification.

These represent some of the more prominent HA-oriented features that become readily apparent when the QNX Neutrino RTOS forms the basis of an HA design.

Chapter 4
Using the High Availability Manager

The High Availability Manager (HAM) provides a mechanism for monitoring processes and services on your system. The goal is to provide a resilient manager (or “smart watchdog”) that can perform multistage recovery when system services or processes fail, do not respond, or provide an unacceptable level of service. The HA framework, including the HAM, uses a simple publish/subscribe mechanism to communicate interesting system events between interested components in the system. By automatically integrating into the native networking mechanism (QNET), this framework transparently extends a local monitoring mechanism to a network.

The HAM acts as a conduit through which the rest of the system can both obtain and deliver information regarding the state of the system as a whole. The system could be a single node or a collection of nodes connected via QNET. The HAM can monitor specific processes and can control the behavior of the system when specific components fail and need to be recovered. The HAM also permits external detectors to report interesting events to the system, and can associate actions with the occurrence of these events.

In many HA systems, single points of failure (SPOFs) must be identified and dealt with carefully. Since the HAM maintains information about the health of the system and also provides the basic recovery framework, the HAM itself must never become a SPOF.

As a self-monitoring manager, the HAM is resilient to internal failures. If, for whatever reason, the HAM itself is stopped abnormally, it can immediately and completely reconstruct its own state. A mirror process called the Guardian perpetually stands ready and waiting to take over the HAM's role. Since all state information is maintained in shared memory, the Guardian can assume the exact same state that the original HAM was in before the failure.

But what happens if the Guardian terminates abnormally? The Guardian (now the new HAM) creates a new Guardian for itself *before taking the place of the original HAM*. Practically speaking, therefore, one can't exist without the other.

Since the HAM/Guardian pair monitor each other, the failure of either one can be completely recovered from. The only way to stop HAM is to explicitly instruct it to terminate the Guardian and then to terminate itself.

HAM hierarchy

HAM consists of three main components:

- *Entities* (p. 24)
- *Conditions* (p. 25)
- *Actions* (p. 27)

Entities

Entities are the fundamental units of observation/monitoring in the system. Essentially, an entity is a process (*pid*). As processes, all entities are uniquely identifiable by their *pid*s. Associated with each entity is a symbolic name that can be used to refer to that specific entity. Again, the names associated with entities are unique across the system. Managers are currently associated with a node, so uniqueness rules apply to a node. As we'll see later, this uniqueness requirement is very similar to the naming scheme used in a hierarchical filesystem.

The basic entity types are:

Self-attached **entities**

These are processes that explicitly choose to be HA-aware. These processes use the *ham_attach_self()* (p. 114) and *ham_detach_self()* (p. 140) functions to connect to and disconnect from a HAM. Self-attached processes are compiled against the HAM API library, and the lifetime of the monitoring is from the time of the *ham_attach_self()* call to the time of the *ham_detach_self()* call.

Self-attached entities can also choose to send heartbeats to a HAM, which will then monitor them for failure. Since arbitrary processes on the system aren't necessarily “trackable” for failure (i.e., they're not in session 1, not child processes, etc.), you can use this heartbeat mechanism to monitor such processes.

Self-attached entities can, on their own, decide at exactly what point in their lifespan they want to be monitored, what conditions they want acted upon, and when they want to stop the monitoring. In other words, this is a situation where a process says, “Do the following if I die.”

Externally attached **entities**

These are generic processes in the system that are being monitored. These could be arbitrary daemons or service providers whose health is deemed important. This method is useful for the case where Process A says, “Tell me when Process B dies” but Process B needn't know about this at all.

Global entity

A global entity is really just a place holder for matching any entity. It can be used to associate actions that will be triggered when an interesting event is detected with respect to any entity on the system. The term global refers to the set of entities being monitored in the system. This permits one to say things like “when any process dies or when any process misses a heartbeat, do the following”. The global entity is never added or removed, but is only referred to. Conditions can be added/removed to the global entity as usual, and actions added/removed from any of the conditions.

To get a handle for a global entity, call *ham_entity_handle()* (p. 153), passing `NULL` for the *ename* argument.

The `dumper` process is normally used to obtain core images of processes that terminate abnormally as a result of performing any illegal operations. A HAM receives notification of such terminations from `dumper`. In addition the HAM also receives notification, from the system, of the termination of any process that is in session 1. This includes daemon processes that call *procmgr_daemon()*, thereby detaching themselves from their controlling terminal.

If a process calls *daemon()*, a new process is created and replaces the original one, becoming the session leader. If the HAM was monitoring the original process, it automatically switches to monitoring the new process instead.

Conditions

Conditions are associated with entities. These conditions represent the state of the entity. Here are some examples of conditions:

Condition	Description
`CONDDEATH`	The entity has died.
`CONDABNORMALDEATH`	The entity has died an abnormal death. This condition is triggered whenever an entity dies by a mechanism that results in the generation of a core file (see `dumper` in the *Utilities Reference* for details).

Condition	Description
`CONDDETACH`	The entity that was being monitored is detaching. This ends HAM's monitoring of that entity.
`CONDATTACH`	An entity for whom a place holder was previously created (someone has subscribed to events relating to this entity), has joined the system. This is also the start of the monitoring of the entity by a HAM.
`CONDHBEATMISSEDHIGH`	The entity missed sending a heartbeat message specified for a condition of "high" severity.
`CONDHBEATMISSEDLOW`	The entity missed sending a heartbeat message specified for a condition of "low" severity.
`CONDRESTART`	The entity was restarted. This condition is true *after* the entity is successfully restarted.
`CONDRAISE`	An externally detected condition is reported to a HAM. Subscribers can associate actions with these externally detected conditions.
`CONDSTATE`	An entity reports a state transition to a HAM. Subscribers can associate actions with specific state transitions.
`CONDANY`	This condition type matches any condition type. It can be used to associate the same actions with one of many conditions.

The conditions described above with the exception of `CONDSTATE`, `CONDRAISE` and `CONDANY` are automatically detected and/or triggered by a HAM (i.e., the HAM is the publisher of the conditions). The `CONDSTATE` and `CONDRAISE` conditions are published to a HAM by external detectors. For all conditions, subscribers can associate with lists of actions that will be performed in sequence when the condition is triggered. Both the `CONDSTATE` and `CONDRAISE` conditions provide filtering capabilities so the subscribers can selectively associate actions with individual conditions, based on the information published.

Conditions are also associated with symbolic names, which also need to be unique within an entity.

The HAM architecture is *extensible*. Several conditions are automatically detected by a HAM. Also, by using the *Condition Raise* mechanism other components in the system can notify a HAM of interesting events in the system. These conditions can be fully customized. Also, by studying the source code, it is possible to add the capability of detecting other conditions into the HAM (e.g., low memory, high CPU utilization, low disk space, etc.) to suit your HA application.

Actions

Actions are associated with conditions. A condition can contain multiple actions. The actions are executed whenever the corresponding condition is true. Actions within a condition execute in FIFO order (the order in which they were added into the condition). Multiple conditions that are true are triggered simultaneously in an arbitrary order. Conditions specified as `HCONDINDEPENDENT` will execute in a separate thread of execution, in parallel with other conditions. (See the section *Condition functions* (p. 39) in this chapter.)

The HAM API includes several functions for different kinds of actions:

Action	Description
ham_action_restart() (p. 104)	This action restarts the entity.
ham_action_execute() (p. 72)	Executes an arbitrary command (e.g., to start a process).
ham_action_notify_pulse() (p. 96)	Notifies some process that this condition has occurred. This notification is sent using a specific *pulse* with a value specified by the process that wished to receive this notify message. Pulses can be delivered to remote nodes, by specifying the appropriate node specifier.
ham_action_notify_signal() (p. 99)	Notifies some process that this condition has occurred. This notification is sent using a specific *realtime signal* with a value specified by the process that wished to receive this notify message. Signals can be delivered to remote nodes, by specifying the appropriate node specifier.

Action	Description
ham_action_notify_pulse_node() (p. 96)	This is the same as the *ham_action_notify_pulse()* described above, except that the node name specified for the recipient of the pulse can be given using the fully qualified node name instead of the node identifier.
ham_action_notify_signal_node() (p. 99)	This is the same as the *ham_action_notify_signal()* described above, except that the node name specified for the recipient of the signal can be given using the fully qualified node name instead of the node identifier.
ham_action_waitfor() (p. 107)	This action lets you insert delays between consecutive actions in a sequence. You can also wait for certain names to appear in the namespace.
ham_action_heartbeat_healthy() (p. 92)	Resets the heartbeat mechanism for an entity that had previously missed sending heartbeats, and had triggered a missed heartbeat condition, but has now recovered.
ham_action_log() (p. 94)	This allows one to insert a customizable verbosity message into the activity log maintained by a HAM.

Actions are also associated with symbolic names, which are unique within a specific condition.

Again, the HAM architecture is extensible, so you may add your own action functions as you see fit.

Action Fail actions

When an action in a list of actions fails, one can specify an alternate list of actions that will be performed to recover from the failure of the given action. These actions are referred to as *action_fail* actions, and are associated with each individual action. The action_fail actions are essentially the same set of actions that would normally be executed with the exception of *ham_action_restart()* and *ham_action_heartbeat_healthy()*). Here's the list of action fail actions:

Action	Description
ham_action_fail_execute() (p. 75)	Executes an arbitrary command (e.g., to start a process).
ham_action_fail_notify_pulse() (p. 80)	Notifies some process that this condition has occurred. This notification is sent using a specific pulse with a value specified by the process that wished to receive this notify message. Pulses can be delivered to remote nodes by specifying the appropriate node specifier.
ham_action_fail_notify_signal() (p. 83)	Notifies some process that this condition has occurred. This notification is sent using a specific realtime signal with a value specified by the process that wished to receive this notify message. Signals can be delivered to remote nodes by specifying the appropriate node specifier.
ham_action_fail_notify_pulse_node() (p. 80)	This is the same as the *ham_action_fail_notify_pulse()* described above, except that the node name specified for the recipient of the pulse can be given using the fully qualified node name instead of the node identifier.
ham_action_fail_notify_signal_node() (p. 83)	This is the same as the *ham_action_fail_notify_signal()* described above, except that the node name specified for the recipient of the signal can be given using the fully qualified node name instead of the node identifier.
ham_action_fail_waitfor() (p. 86)	This action lets you insert delays between consecutive actions in a sequence. You can also wait for certain names to appear in the namespace.
ham_action_fail_log() (p. 78)	This allows one to insert a customizable verbosity message into the activity log maintained by a HAM.

Multistaged recovery

This complete mechanism allows us to perform recovery of a failure of a single service or process in a multi-staged fashion.

For example, suppose you've started `fs-nfs2` (the NFS filesystem) and then mounted a few directories from multiple sources. You can instruct HAM to restart `fs-nfs2` upon failure, and also to remount the appropriate directories as required after restarting the NFS process. And if during the lifespan of `fs-nfs2` some directories are unmounted, you can remove those particular actions from the set of actions to be performed.

As another example, suppose `io-pkt*` (network I/O manager) were to die. We can tell a HAM to restart it and also to load the appropriate network drivers (and maybe a few more services that essentially depend on network services in order to function).

State of the HAM

Effectively, a HAM's internal state is like a hierarchical filesystem, where entities are like directories, conditions associated with those entities are like subdirectories, and actions inside those conditions are like leaf nodes of this tree structure.

A HAM also presents this state as a read-only filesystem under `/proc/ham`. As a result, arbitrary processes can also view the current state (e.g., you can do `ls /proc/ham`).

Besides presenting a view of the state as a filesystem, for each item (entity/condition/action) a HAM can also display statistics and information relating to it in a corresponding `.info` file at each level in a HAM filesystem under `/proc/ham`.

Example of the view shown in `/proc/ham`

Consider the following simple example where a HAM is monitoring `inetd` and restarts it when it dies:

```
# ls -al /proc/ham
total 2
-r--------  1 root      root            175 Aug 30 23:05 .info
dr-x------  1 root      root              1 Aug 30 23:06 inetd
```

The `.info` file at the highest level provides information about the HAM and the Guardian, as well as an overview of the entities and other objects in the system:

```
# cat /proc/ham/.info
Ham Pid            : 10993674
Guardian Pid       : 10997782
Ham Failures       : 0
Guardian Failures  : 0
Num Entities       : 1
Num Conditions     : 1
Num Actions        : 1
```

In this case the only entity being monitored is `inetd`, which appears as a directory at the top level under `/proc/ham`:

```
# ls -al /proc/ham/inetd
total 2
-r--------  1 root      root            173 Aug 30 23:06 .info
dr-x------  1 root      root              1 Aug 30 23:06 death

# cat /proc/ham/inetd/.info
Path            : inetd
Entity Pid      : 11014167
Num conditions  : 1
Entity type     : ATTACHED
Stats:
Created         : 2001/08/30 23:04:49:930148650
Num Restarts    : 0
```

As you can see, the `.info` provides information and statistics relating to the `inetd` entity that is being monitored. The information is generated dynamically and contains up-to-date data for each entity.

The `inetd` entity has associated with it only one *condition* (i.e., death), which is triggered when the entity dies.

```
# ls -al /proc/ham/inetd/death
total 2
-r--------  1 root      root            126 Aug 30 23:07 .info
-r--------  1 root      root            108 Aug 30 23:07 restart

# cat /proc/ham/inetd/death/.info
Path             : inetd/death
Entity Pid       : 11014167
Num Actions      : 1
Condition ReArm  : ON
Condition type   : CONDDEATH
```

Similarly, there's only one *action* associated with this death condition: the *restart* mechanism. Each action under the condition appears as a file under the appropriate condition directory. The file contains details about the action that will be performed when the condition is triggered.

```
# cat /proc/ham/inetd/death/restart
Path         : inetd/death/restart
Entity Pid   : 11014167
Action ReArm : ON
Restart Line : /usr/sbin/inetd -D
```

If `inetd` isn't a self-attached entity, you need to specify the `-D` option to it, to force `inetd` to daemonize by calling *procmgr_daemon()* instead of by calling *daemon()*. The HAM can see death messages only from self-attached entities, processes that terminate abnormally, and tasks that are running in session 1, and the call to *daemon()* doesn't put the caller into that session.

If `inetd` is a self-attached entity, you don't need to specify the `-D` option because the HAM automatically switches to monitoring the new process that *daemon()* creates.

When `inetd` dies, all the actions associated with a death condition under it are executed:

```
# slay inetd

# cat /proc/ham/inetd/.info
Path             : inetd
Entity Pid       : 11071511  <- new pid of entity
Num conditions   : 1
Entity type      : ATTACHED
Stats:
Created          : 2001/08/30 23:04:49:930148650
Last Death       : 2001/08/30 23:10:31:889820814
Restarted        : 2001/08/30 23:10:31:904818519
Num Restarts     : 1
```

As you can see, the statistics relating to the entity `inetd` are updated.

Similarly, if a HAM itself is terminated, the Guardian takes over as the new HAM, and creates a Guardian for itself.

```
# cat /proc/ham/.info
Ham Pid             : 10993674  <----- This is the HAM
Guardian Pid        : 10997782  <----- This is the Guardian
Ham Failures        : 0
```

```
Guardian Failures  : 0
Num Entities       : 1
Num Conditions     : 1
Num Actions        : 1

... Kill the ham ....

# /bin/kill -9 10993674        <---- Simulate failure

... re-read the stats ...

# cat /proc/ham/.info
Ham Pid            : 10997782  <----- This is the new HAM
Guardian Pid       : 11124746  <----- This is the Guardian
Ham Failures       : 1
Guardian Failures  : 0
Num Entities       : 1
Num Conditions     : 1
Num Actions        : 1
```

As you can see, the old Guardian is now the new HAM, and a new Guardian has been created. All entities and conditions remain as before; the monitoring continues as usual. The HAM and the Guardian ignore all signals that they can.

HAM API

A HAM provides an API for you to use in order to interact with it. This API provides a collection of functions to:

- connect to and disconnect from a HAM
- add entities/conditions/actions to (and remove them from) the set of things currently being monitored.

The API is implemented as a library that you can link against. The library is thread-safe and also cancellation-safe.

Connect/disconnect functions

The HAM API library maintains only one connection to the HAM. The library itself is thread-safe, and multiple connections (from different threads) or the same thread are multiplexed on the same single connection to a HAM. The library maintains reference counts.

Here are the basic connect functions:

```
/* Basic connect functions
   return success (0) or failure (-1, with errno set) */

int ham_connect(unsigned flags);
int ham_connect_nd(int nd, unsigned flags);
int ham_connect_node(const char *nodename, unsigned flags);

int ham_disconnect(unsigned flags);
int ham_disconnect_nd(int nd, unsigned flags);
int ham_disconnect_node(const char *nodename, unsigned flags);
```

These functions are used to open or close connections to a HAM. The first call to *ham_connect*()* will open the *fd*, while subsequent calls will increment the reference count.

Similarly, *ham_disconnect()* (p. 142) will decrement the count until zero; the call that makes the count zero will close the *fd*. The functions return -1 on error, and 0 on success. Similarly *ham_disconnect*()* will decrement the reference count until zero, with the call that makes the count zero closing the *fd*. The functions return -1 on error with errno set, and 0 on success.

In a multithreaded situation, there will exist only one open connection to a given HAM at any given time, even if multiple threads were to perform *ham_connect*()*/*ham_disconnect*()* calls.

The *ham_*_nd()* and *ham_*_node()* versions of the calls are used to open a connection to a remote HAM across QNET. The *nd* that is passed to the function is the node identifier that refers to the remote host at the instant the call is made. Since node identifiers are transient values, it is essential that the node identifier is obtained just prior to the call. The other option is to use the fully qualified node name (FQNN) of the host and to pass this as the *nodename* parameter. An *nd* of ND_LOCAL_NODE (a constant defined in sys/netmgr.h) or a *nodename* of NULL (or the empty string) are equivalent, and refer to the current node. (This is also the same as calling *ham_connect()* or *ham_disconnect()* directly).

Calls to *ham_connect()*, *ham_connect_nd()*, and *ham_connect_node()* can be freely mixed, as long as the number of connect calls equals the number of disconnect calls for each connection to a specific (local or remote) HAM before the connection (*fd*) is closed.

Attach/detach functions

For self-attached entities

```
ham_entity_t *ham_attach_self(char *ename, uint64_t hp, int hpdl,
                 int hpdh, unsigned flags);
int ham_detach_self(ham_entity_t *ehdl, unsigned flags);
```

You use these two functions to attach/detach a process to/from a HAM as a self-attached entity.

The *ename* argument represents the symbolic name for this entity, which needs to be unique in the system (of all monitored entities at the instant the call is made).

The *hp* argument represents time values in nanoseconds for the *heartbeat period*. Heartbeating can be used to ensure “liveness” of the monitored entity. Liveness is a property that describes a component's useful progress. In many cases, the availability of a system component is compromised not because the component has necessarily died, but because it isn't responding or making any progress. The heartbeating mechanism lets you specify that a component will issue a heartbeat at a given interval, and if it misses a certain number of heartbeats, then that would constitute a heartbeat-missed condition.

The *hpdl* and *hpdh* represent the number of heartbeats that can be missed before the conditions *heartbeatmissedlow* and *heartbeatmissedhigh* are triggered. The HAM API library registers this request with a HAM and also creates a thread that keeps the connection to a HAM open. If the entity were to abnormally terminate, the connection to the HAM is closed, and the HAM will know that this is an abnormal termination (since *ham_detach_self()* (p. 140) wasn't called first).

On the other hand, if a HAM were to abnormally fail (extremely unlikely) and the Guardian takes over as the new HAM, the connection to the old HAM will have gone stale. In that case, the Guardian notifies all self-attached entities to reattach. The extra thread mentioned above handles this reattach transparently.

If a connection to a HAM is already open, then *ham_attach_self()* (p. 114) uses the same connection, but increments the reference count of connections opened by this client. A client that indicates that it will heartbeat at a certain period must call *ham_heartbeat()* (p. 157) to actually transmit a heartbeat to the HAM.

The library also verifies whether the *ename* provided by the caller is unique. If it doesn't already exist, then this request is forwarded to a HAM, which also checks it again to avoid any race conditions in creating new entities. The *ham_attach_self()* (p. 114) returns a generic handle, which can be used to detach the process from the HAM later. Note that this handle is an opaque pointer that's also used to add conditions and actions as shown below.

The *ham_detach_self()* (p. 140) function is used to close the connection to a HAM. From this point on, the HAM will no longer monitor this process as a self-attached entity. The extra thread is canceled. The *ham_detach_self()* function takes as an argument the handle returned by *ham_attach_self()*.

Code snippet using self-attach/detach calls

The following snippet of code uses the *ham_attach/detach_self()* functions:

```
...
ham_entity_t *ehdl; /* The entity Handle */
int status;

/*
 connects to a HAM with a heartbeat of 5 seconds
 and an entity name of "client1", and no flags
 it also specifies hpdh = 4, and hpdh = 8
*/

ehdl = ham_attach_self("client1", 5000000000, 4, 8, 0);
if (ehdl == NULL) {
  printf("Could not attach to Ham\n");
  exit(-1);
}
/* Detach from a HAM using the original handle */
status = ham_detach_self(ehdl,0);
...
```

For attaching/detaching all other entities

```
ham_entity_t *ham_attach(char *ename, int nd, pid_t pid, char *line,
              unsigned flags);
ham_entity_t *ham_attach_node(char *ename, const char *nodename, pid_t pid,
              char *line, unsigned flags);
int ham_detach(ham_entity_t *ehdl, unsigned flags);
int ham_detach_name(int nd, char *ename, unsigned flags);
int ham_detach_name_node(const char *nodename, char *ename, unsigned flags);
```

These attach/detach/detach-name functions are very similar to the **_self()* functions above, except here the calling process asks a HAM to monitor a different process.

This mechanism allows for arbitrary monitoring of entities that already exist and aren't compiled against the HAM API library. In fact, the entities that are being monitored needn't even be aware that they're being monitored.

You can use the *ham_attach()* (p. 110) call either to:

- start an entity and continue to monitor it or:

- begin monitoring an entity that's already running.

In the *ham_attach()* call, if *pid* is -1, then we assume that the entity isn't running. The entity is started now using *line* as the startup command line for it. But if *pid* is greater than 0, then *line* is ignored and the *pid* given is attached to as an entity. Again *ename* needs to be unique across all entities currently registered.

The *nd* specifier in *ham_attach()* and *ham_detach_name()*, and the *nodename* specifier in the *ham_attach_node()* and *ham_detach_name_node()* versions of the calls are used to refer to a remote HAM across Qnet. The *nd* that is passed to the function is the node identifier that refers to the remote host at the instant the call is made. Since node identifiers are transient values, it is essential that the node identifier is obtained just prior to the call. The other option is to use the fully qualified node name (FQNN) of the host and to pass this as the *nodename* parameter. An *nd* of `ND_LOCAL_NODE` (a constant defined in `sys/netmgr.h` or a *nodename* of `NULL` (or the empty string) are equivalent, and refer to the current node.

The *ham_detach*()* functions stop monitoring a given entity. The *ham_detach()* (p. 136) call takes as an argument the original handle returned by *ham_attach()*. You can also call *ham_detach_name()* (p. 138), which uses the entity's name instead of the handle.

Note that the entity handle can also be used later to add *conditions* to the entity (described below).

Code snippet using attach/detach calls

```
...
ham_entity_t *ehdl;
int status;
ehdl = ham_attach("inetd", 0, -1, "/usr/sbin/inetd -D", 0);
/* inetd is started, running and monitored now */
...
...
status = ham_detach(ehdl,0);
...
...
```

Of course the attach and detach needn't necessarily be performed by the same caller:

```
...
ham_entity_t *ehdl;
int status;
/* starts and begins monitoring inetd */
ehdl = ham_attach("inetd", 0, -1, "/usr/sbin/inetd -D", 0);
...
...
/* disconnect from Ham (monitoring still continues) */
exit(0);
```

And to detach `inetd`:

```
...
int status;
/* stops monitoring inetd. */
status = ham_detach_name(0, "inetd", 0);
```

```
...
exit(0);
```

If `inetd` were already running, say with *pid* `105328676`, then we can write the attach/detach code as follows:

```
ham_entity_t *ehdl;
int status;
ehdl = ham_attach("inetd", 0, 105328676, NULL, 0);
...
...
status = ham_detach(ehdl,0);
/* status = ham_detach_name(0, "inetd",0); */
...
...
exit(0);
```

For convenience, the *ham_attach()* and *ham_detach()* functions connect to a HAM if such a connection doesn't already exist. We do this only to make the use of the functions easier.

The connections to a HAM persist only for the duration of the attach/detach calls; any subsequent requests to the HAM must be preceded by the appropriate *ham_connect()* calls.

The best way to perform a large sequence of requests to a HAM is to:

1. Call *ham_connect()* (p. 134) before the first request.
2. Call *ham_disconnect()* (p. 142) after the last request.

This is the most efficient method, because it guarantees that there's always the same connection open to the HAM.

Entity functions

The *ham_attach_*()* functions are normally used when an entity is either already running or will be started by a HAM, and monitoring begins with the invocation of the *ham_attach*()* call. The HAM API also provides two functions that allow users to create placeholders for entities that are not yet running and that might be started in the future. This allows subscribers of interesting events to indicate their interest in these events, without necessarily waiting for a publisher (other entity/HAM) to create the entity.

```
ham_entity_t *ham_entity(const char *ename, int nd, unsigned flags);
ham_entity_t *ham_entity_node(const char *ename, const char *nodename,
               unsigned flags);
```

These functions create entity place holders with the name specified *ename*, on the corresponding node described by either the node identifier *nd* or the nodename given by *nodename*. Once created, these placeholders can be used to add conditions and actions to their associated entities. When a subsequent *ham_attach*()* call is made that references the same *ename*, it will fill the entity place holder with the appropriate process ID. From that time onwards, the entity is monitored normally.

Condition functions

```
ham_condition_t *ham_condition(ham_entity_t *ehdl, int type,
                 const char *cname, unsigned flags);
int ham_condition_remove(ham_condition_t *chdl, unsigned flags);
```

Each entity can be associated with various conditions. And for each of these conditions there's a set of actions that will be performed in sequence when the condition is true. If an entity has multiple conditions that are true simultaneously with different sets of actions associated with each condition, then all the actions are performed for each condition, in sequence.

This mechanism lets you combine actions together into sets and choose to remove/control them as a single “group” instead of as individual items.

Since conditions are associated with entities, an *entity handle* must be available in order to add conditions. The *ham_condition*()* functions return an opaque pointer that is a condition handle, which you can use later to either remove a condition or add actions to the condition.

Condition types

You can specify any of the following for *type*:

CONDDEATH

The entity has died.

CONDABNORMALDEATH

The entity has died an abnormal death. This condition is triggered whenever an entity dies by a mechanism that results in the generation of a core file (see `dumper` in the *Utilities Reference* for details).

CONDDETACH

The entity that was being monitored is detaching. This ends HAM's monitoring of that entity.

CONDATTACH

An entity for whom a place holder was previously created (someone has subscribed to events relating to this entity), has joined the system. This is also the start of the monitoring of the entity by a HAM.

CONDHBEATMISSEDHIGH

The entity missed sending a heartbeat message specified for a condition of “high” severity.

CONDHBEATMISSEDLOW

The entity missed sending a heartbeat message specified for a condition of "low" severity.

CONDRESTART

The entity was restarted. This condition is true *after* the entity is successfully restarted.

CONDANY

This condition type matches any condition type. It can be used to associate the same actions with one of many conditions.

The CONDATTACH, CONDDETACH and CONDRESTART conditions are triggered by the HAM, when entities attach, detach, or restart respectively. The CONDHBEATMISSEDHIGH and CONDHBEATMISSEDLOW conditions are triggered internally by the HAM when it detects the missed heartbeat conditions, as defined by the entities when they indicated their original intent to heartbeat.

CONDDEATH is triggered whenever an entity dies. CONDABNORMALDEATH is triggered only when an abnormal death takes place, but such an abnormal death also triggers a CONDDEATH condition.

You use the *detach* condition to perform some actions whenever a monitored entity properly detaches from a HAM. After this point, the HAM will no longer monitor the entity. In effect, you can use this to "notify" interested clients when the HAM can no longer provide any more information about the detaching entity.

The *restart* condition is asserted and triggered by a HAM automatically if an entity dies and is restarted.

Condition flags

HCONDNOWAIT

Guarantees that there can be no "waitfor" statements in the list of actions in this condition. All conditions that are flagged HCONDNOWAIT are handled in a separate thread, and thus aren't delayed in any way by the nature of the actions in other conditions.

HCONDINDEPENDENT

If this flag is set, then all actions in this condition are executed in a separate thread. This lets you insert delays into a condition, without incurring any delays in other conditions.

If a condition is flagged with both HCONDINDEPENDENT and HCONDNOWAIT, then HCONDNOWAIT takes precedence, and all actions in this condition are executed in the same thread as *all* other conditions that are also flagged as HCONDNOWAIT. This is because all HCONDNOWAIT conditions are guaranteed to have minimal delays already.

If a condition is flagged with neither HCONDNOWAIT nor HCONDINDEPENDENT, it is treated as an OTHER condition, implying that it will be executed in the FIFO order among all conditions that are true.

To sum up:

1. Whenever a condition (e.g., CONDDEATH, CONDDETACH, etc.) occurs, all conditions flagged HCONDNOWAIT are executed in FIFO order in a single thread.
2. All conditions flagged HCONDINDEPENDENT (but not HCONDNOWAIT) are executed each in a separate thread.
3. All other conditions are executed in FIFO order in one single thread.

This limits the number of threads in all to be at most:

(number of HCONDINDEPENDENT conditions) + 2

That is, one for all the conditions flagged HCONDNOWAIT, and one for all OTHER conditions.

In addition, within a condition, all actions are also executed in FIFO order. This is true irrespective of whether the conditions are HCONDNOWAIT or HCONDINDEPENDENT.

Action functions

```
/* action operations             */
ham_action_t *ham_action_restart(ham_condition_t *chdl, const char *aname,
              const char *path, unsigned flags);
ham_action_t *ham_action_execute(ham_condition_t *chdl, const char *aname,
              const char *path, unsigned flags);
ham_action_t *ham_action_waitfor(ham_condition_t *chdl, const char *aname,
              const char *path, int delay, unsigned flags);
ham_action_t *ham_action_notify_pulse(ham_condition_t *chdl, const char *aname,
              int nd, int topid, int chid, int pulsecode, int value,
              unsigned flags);
ham_action_t *ham_action_notify_signal(ham_condition_t *chdl, const char *aname,
              int nd, pid_t topid, int signum, int code, int value,
              unsigned flags);
ham_action_t *ham_action_notify_pulse_node(ham_condition_t *chdl,
              const char *aname, const char *nodename, int topid, int chid,
              int pulsecode, int value, unsigned flags);
ham_action_t *ham_action_notify_signal_node(ham_condition_t *chdl,
              const char *aname, const char *nodename, pid_t topid,
              int signum, int code, int value, unsigned flags);
ham_action_t *ham_action_heartbeat_healthy(ham_condition_t *chdl,
              const char *aname, unsigned flags);
ham_action_t *ham_action_log(ham_condition_t *chdl, const char *aname,
              const char *msg, unsigned attachprefix, int verbosity,
              unsigned flags);

/* remove an action              */
int ham_action_remove(ham_action_t *ahdl, unsigned flags);
```

As mentioned earlier, a HAM currently supports several different types of *action functions*, but note that you can add your own action functions to suit your particular HA application.

ham_action_restart() **(p. 104)**

Provides a restart mechanism for the entity in the event that a *death* condition has occurred. This implies that the entity in question has terminated; the *restart* action will restart the entity and also keep track of the new *pid* that the entity will now be associated with.

Restart actions can be associated only with *death* conditions. And across all conditions of type *death*, there can be only a single *restart* action at any time. This ensures that the entity is restarted only if it terminates, and only once. (Conditions of type *death* include conditions of the types `CONDDEATH` and `CONDABNORMALDEATH`.

[*ham_action_execute()*](#) (p. 72)

Executes an arbitrary command in the event that the condition is true. This could be any executable command line. When the condition in question is true, the list of actions is traversed and executed in sequence.

This executes a command line as specified in the parameters. The command line must contain the FULL path to the executable along with all parameters to be passed to it. The command line is in turn passed onto a *spawn* command by a HAM to create a new process that will execute the command.

You'll find *execute* actions useful when you need to set up a multistage recovery. For example, if `fs-nfs2` dies and is restarted, the *ham_action_execute()* function lets you remount any directories that are required after `fs-nfs2` is restarted.

You can have an *execute* action take place immediately by setting the `HACTIONDONOW` flag. Again, this is useful in startup situations when an entity is created in many stages.

Note that `HACTIONDONOW` is ignored for *waitfor* actions. So in order to insert delays into a sequence of actions flagged `HACTIONDONOW`, you'll need to insert the delays in the client program (between calls to *ham_action*()*).

[*ham_action_waitfor()*](#) (p. 107)

Given a sequence of actions in a condition that will execute in FIFO order, you can insert delays into the execution sequence by using *ham_action_waitfor()* (as long as the condition permits it — see the section Condition functions (p. 39) in this chapter). The delay specified is in multiples of 100 msecs.

The *ham_action_waitfor()* call takes as an argument a *path* component, which can be used to wait for a specific name to appear in the name space. If *path* is NULL, the waitfor is for exactly *delay* msecs. But if *path* is specified, the waitfor is for either *delay* msecs or until *path* appears in the namespace, whichever occurs earlier. Note that the delay when a pathname is specified is in integral multiples of 100 msecs.

If a pathname is specified, the delays will be the closest integral multiple of 100 msecs, rounding up. A delay of `0` effectively disables the waitfor, making the *pathname* specification redundant.

ham_action_notify_pulse() **(p. 96),** ***ham_action_notify_signal()*** **(p. 99)**

The *ham_action_notify_pulse()* function sends the appropriate pulse to the given *nd*/*pid*/*chid*.

The *action_notify_signal()* sends an appropriate realtime signal with a value to the *pid* that requests it.

Actions can persist across a restart if the entity is restarted. Similarly, conditions can also be set to persist (i.e., you can rearm them) after a restart of the entity. You can do this by ORing `HREARMAFTERRESTART` into the flags argument to either the *ham_condition()* call or to the appropriate action statement.

If a condition persists when an entity is restarted, each individual action is checked to see if it also persists. Actions that needn't be rearmed are performed once and removed. Any actions that fail are also removed, even if they're set to be rearmed.

If a condition isn't marked as *rearmed*, then all actions under it are automatically removed, since the actions are associated only with the condition and can't be retained if the condition no longer exists.

The persistence of conditions and actions across a restart depends on the restart of the entity itself. So if the entity isn't restarted (i.e., there's no `ACTIONRESTART` or the `ACTIONRESTART` fails for some reason), then the entity is removed, along with all conditions and actions associated with the entity as well.

ham_action_notify_pulse_node() **(p. 96)**

This is the same as the *ham_action_notify_pulse()* above, except that the node name specified for the recipient of the pulse can be given using the fully qualified node name instead of the node identifier (*nd*).

ham_action_notify_signal_node() **(p. 99)**

This is the same as the *ham_action_notify_signal()* above, except that the node name specified for the recipient of the signal can be given using the fully qualified node name instead of the node identifier (*nd*).

Action fail functions

```
/* action fail operations          */
int ham_action_fail_execute(ham_action_t *ahdl, const char *aname,
    const char *path, unsigned flags);
int ham_action_fail_waitfor(ham_action_t *ahdl, const char *aname,
    const char *path, int delay, unsigned flags);
int ham_action_fail_notify_pulse(ham_action_t *ahdl, const char *aname,
    int nd, int topid, int chid, int pulsecode, int value, unsigned flags);
int ham_action_fail_notify_signal(ham_action_t *ahdl, const char *aname,
    int nd, pid_t topid, int signum, int code, int value, unsigned flags);
int ham_action_fail_notify_pulse_node(ham_action_t *ahdl, const char *aname,
    const char *nodename, int topid, int chid, int pulsecode, int value,
    unsigned flags);
int ham_action_fail_notify_signal_node(ham_action_t *ahdl, const char *aname,
    const char *nodename, pid_t topid, int signum, int code, int value,
    unsigned flags);
int ham_action_fail_log(ham_action_t *ahdl, const char *aname,
    const char *message, unsigned attachprefix, int verbosity, unsigned flags);

/* remove an action fail operation */
int ham_action_fail_remove(ham_action_t *ahdl, const char *aname,
    unsigned flags);
```

These actions are used to associate a list of actions that will be executed when an action in a condition fails. These functions are similar to the corresponding action functions described in the previous section, the primary difference being the first parameter, which in the case of these functions is a handle to an action (as opposed to a handle to a condition).

Example to monitor `inetd`

The following code snippet shows how to begin monitoring the `inetd` process:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/netmgr.h>
#include <fcntl.h>
#include <ha/ham.h>

int main(int argc, char *argv[])
{
  int status;
  char *inetdpath;
    ham_entity_t *ehdl;
    ham_condition_t *chdl;
    ham_action_t *ahdl;
    int inetdpid;

    inetdpath = strdup("/usr/sbin/inetd -D");
    inetdpid = -1;
    ham_connect(0);
    ehdl = ham_attach("inetd", ND_LOCAL_NODE, inetdpid, inetdpath, 0);
    if (ehdl != NULL)
    {
      chdl = ham_condition(ehdl,CONDDEATH, "death", HREARMAFTERRESTART);
    if (chdl != NULL) {
        ahdl = ham_action_restart(chdl, "restart", inetdpath,
                              HREARMAFTERRESTART);
          if (ahdl == NULL)
              printf("add action failed\n");
        }
        else
            printf("add condition failed\n");
    }
    else
        printf("add entity failed\n");
    ham_disconnect(0);
    exit(0);
}
```

Example to monitor `fs-nfs2`

The following code snippet shows how to begin monitoring the `fs-nfs2` process:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/netmgr.h>
#include <fcntl.h>
#include <ha/ham.h>

int main(int argc, char *argv[])
{
  int status;
    ham_entity_t *ehdl;
    ham_condition_t *chdl;
    ham_action_t *ahdl;
    char *fsnfspath;
    int fsnfs2pid;

    fsnfspath = strdup("/usr/sbin/fs-nfs2");
    fsnfs2pid = -1;

    ham_connect(0);
    ehdl = ham_attach("Fs-nfs2", ND_LOCAL_NODE, fsnfs2pid, fsnfspath, 0);
    if (ehdl != NULL)
    {
      chdl = ham_condition(ehdl,CONDDEATH, "Death", HREARMAFTERRESTART);
    if (chdl != NULL) {
        ahdl = ham_action_restart(chdl, "Restart", fsnfspath,
                              HREARMAFTERRESTART);
          if (ahdl == NULL)
              printf("add action failed\n");
            else {
          ahdl = ham_action_waitfor(chdl, "Delay1", NULL, 2000,
                                  HREARMAFTERRESTART);
            if (ahdl == NULL)
                printf("add action failed\n");
          ahdl = ham_action_execute(chdl, "MountDir1",
                      "/bin/mount -t nfs a.b.c.d:/dir1 /dir1",
                      HREARMAFTERRESTART|HACTIONDONOW));
            if (ahdl == NULL)
                printf("add action failed\n");
          ahdl = ham_action_waitfor(chdl, "Delay2", NULL, 2000,
                     HREARMAFTERRESTART);
            if (ahdl == NULL)
                printf("add action failed\n");
          ahdl = ham_action_execute(chdl, "Mountdir2",
                                  "/bin/mount -t nfs a.b.c.d:/dir2 /dir2",
                                  HREARMAFTERRESTART|HACTIONDONOW);
            if (ahdl == NULL)
                printf("add action failed\n");
            }
        }
        else
            printf("add condition failed\n");
    }
    else
        printf("add entity failed\n");
    ham_disconnect(0);
    exit(0);
}
```

Functions to operate on handles

```
/* Get/Free handles    */
ham_entity_t *ham_entity_handle(int nd, const char *ename, unsigned flags);
ham_condition_t *ham_condition_handle(int nd, const char *ename,
                  const char *cname, unsigned flags);
ham_action_t *ham_action_handle(int nd, const char *ename, const char *cname,
              const char *aname, unsigned flags);
ham_entity_t *ham_entity_handle_node(const char *nodename, const char *ename,
              unsigned flags);
ham_condition_t *ham_condition_handle_node(const char * nodename,
              const char *ename, const char *cname, unsigned flags);
ham_action_t *ham_action_handle_node(const char * nodename, const char *ename,
              const char *cname, const char *aname, unsigned flags);
int ham_entity_handle_free(ham_entity_t *ehdl);
int ham_condition_handle_free(ham_condition_t *chdl);
int ham_action_handle_free(ham_action_t *ahdl);
```

You use the handle functions to get/free handles based on entity, condition, and action names. You can then use these handles later to add or remove conditions and actions. As for all the other functions the **_node*()* variations are used to refer to a HAM that is not necessarily local, using a fully qualified node name (FQNN).

A client example

Here's an example of a client that obtains notifications via pulses and signals about significant events from a HAM. It registers a pulse-notification scheme in the event that `inetd` dies or detaches. It also registers a signal-notification mechanism for the death of `fs-nfs2`.

This example also demonstrates how the delayed notification occurs, and shows how to overcome this using an `HCONDINDEPENDENT` condition.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/netmgr.h>
#include <signal.h>
#include <ha/ham.h>

#define PCODEINETDDEATH      _PULSE_CODE_MINAVAIL+1
#define PCODEINETDDETACH     _PULSE_CODE_MINAVAIL+2
#define PCODENFSDELAYED      _PULSE_CODE_MINAVAIL+3
#define PCODEINETDRESTART1   _PULSE_CODE_MINAVAIL+4
#define PCODEINETDRESTART2   _PULSE_CODE_MINAVAIL+5

#define MYSIG SIGRTMIN+1

int fsnfs_value;

/* Signal handler to handle the death notify of fs-nfs2 */
void MySigHandler(int signo, siginfo_t *info, void *extra)
{
  printf("Received signal %d, with code = %d, value %d\n",
        signo, info->si_code, info->si_value.sival_int);
  if (info->si_value.sival_int == fsnfs_value)
    printf("FS-nfs2 died, this is the notify signal\n");
  return;
}

int main(int argc, char *argv[])
{
  int chid, coid, rcvid;
  struct _pulse pulse;
  pid_t pid;
  int status;
  int value;
  ham_entity_t *ehdl;
  ham_condition_t *chdl;
  ham_action_t *ahdl;
  struct sigaction sa;
  int scode;
  int svalue;

  /* we need a channel to receive the pulse notification on */
  chid = ChannelCreate( 0 );

  /* and we need a connection to that channel for the pulse to be
     delivered on */
  coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );

  /* fill in the event structure for a pulse */
  pid = getpid();
  value = 13;
  ham_connect(0);
  /* Assumes there is already an entity by the name "inetd" */
  chdl = ham_condition_handle(ND_LOCAL_NODE, "inetd","death",0);
```

```
ahdl = ham_action_notify_pulse(chdl, "notifypulsedeath",ND_LOCAL_NODE, pid,
           chid, PCODEINETDDEATH, value, HREARMAFTERRESTART);

ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);

ehdl = ham_entity_handle(ND_LOCAL_NODE, "inetd", 0);
chdl = ham_condition(ehdl, CONDDETACH, "detach", HREARMAFTERRESTART);
ahdl = ham_action_notify_pulse(chdl, "notifypulsedetach",ND_LOCAL_NODE, pid,
         chid, PCODEINETDDETACH, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ham_entity_handle_free(ehdl);

fsnfs_value = 18; /* value we expect when fs-nfs dies */
scode = 0;
svalue = fsnfs_value;
sa.sa_sigaction = MySigHandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_SIGINFO;
sigaction(MYSIG, &sa, NULL);

/*
 Assumes there is an entity by the name "Fs-nfs2".
 We use "Fs-nfs2" to symbolically represent the entity
 fs-nfs2. Any name can be used to represent the
 entity, but it's best to use a readable and meaningful name.
*/
ehdl = ham_entity_handle(ND_LOCAL_NODE, "Fs-nfs2", 0);

/*
 Add a new condition, which will be an "independent" condition.
 This means that notifications/actions inside this condition
 are not affected by "waitfor" delays in other action
 sequence threads
*/
chdl = ham_condition(ehdl,CONDDEATH, "DeathSep",
                  HCONDINDEPENDENT|HREARMAFTERRESTART);
ahdl = ham_action_notify_signal(chdl, "notifysignaldeath",ND_LOCAL_NODE,
                  pid, MYSIG, scode, svalue, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ham_entity_handle_free(ehdl);

chdl = ham_condition_handle(ND_LOCAL_NODE, "Fs-nfs2","Death",0);
/*
 This action is added to a condition that does not
 have an HCONDNOWAIT. Since we are unaware what the condition
 already contains, we might end up getting a delayed notification
 since the action sequence might have "arbitrary" delays and
 "waits" in it.
*/
ahdl = ham_action_notify_pulse(chdl, "delayednfsdeathpulse", ND_LOCAL_NODE,
           pid, chid, PCODENFSDELAYED, value, HREARMAFTERRESTART);

ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);

ehdl = ham_entity_handle(ND_LOCAL_NODE, "inetd", 0);

/* We force this condition to be independent of all others. */
chdl = ham_condition(ehdl, CONDRESTART, "restart",
                         HREARMAFTERRESTART|HCONDINDEPENDENT);
ahdl = ham_action_notify_pulse(chdl, "notifyrestart_imm", ND_LOCAL_NODE,
                  pid, chid, PCODEINETDRESTART1, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ahdl = ham_action_waitfor(chdl, "delay",NULL,6532, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ahdl = ham_action_notify_pulse(chdl, "notifyrestart_delayed", ND_LOCAL_NODE,
                  pid, chid, PCODEINETDRESTART2, value, HREARMAFTERRESTART);

ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ham_entity_handle_free(ehdl);

while (1) {
  rcvid = MsgReceivePulse( chid, &pulse, sizeof( pulse ), NULL );
  if (rcvid < 0) {
    if (errno != EINTR) {
      exit(-1);
      }
    }
    else {
            switch (pulse.code) {
                case PCODEINETDDEATH:
                      printf("Inetd Death Pulse\n");
                    break;
                case PCODENFSDELAYED:
                      printf("Fs-nfs2 died: this is the possibly delayed pulse\n");
                      break;
                case PCODEINETDDETACH:
                      printf("Inetd detached, so quitting\n");
                     goto the_end;
                case PCODEINETDRESTART1:
                      printf("Inetd Restart Pulse: Immediate\n");
                    break;
                case PCODEINETDRESTART2:
                      printf("Inetd Restart Pulse: Delayed\n");
                    break;
              }
    }
  }
  /*
   At this point we are no longer waiting for the
   information about inetd, since we know that it
   has exited.
   We will still continue to obtain information about the
   death of fs-nfs2, since we did not remove those actions.
   If we exit now, the next time those actions are executed
   they will fail (notifications fail if the receiver does not
   exist anymore), and they will automatically get removed and
   cleaned up.
  */
the_end:
  ham_disconnect(0);
  exit(0);
}
```

Note that the HAM API has certain restrictions:

- The names of entities, conditions, and actions (*ename*, *cname*, and *aname*) must not contain a "/" character.
- All names are subject to the length restriction imposed by _POSIX_PATH_MAX (as defined in <limits.h>). Since the names are manifested inside the namespace, the effective length of a name is the maximum length of the name as a path component. In other words, the *combined length* of an entity/condition/action name — including the /proc/ham prefix — must not exceed _POSIX_PATH_MAX.

Starting and stopping a HAM

You start a HAM by running the **ham** utility at the command line:

```
ham
```

The **ham** utility has these command-line options:

-?|h

Display usage message

-d

Disable internal verbosity.

-f

Log verbose output to a file (default is *stderr*).

-t none|relative|absolute|shortabs

Specify the timestamping method. The default is `relative`.

-v

Set verbosity level — extra `-v`'s increase verbosity.

-Vn

Set verbosity level — use a number to specify the level (e.g., `-V3`).

When a HAM starts, it also starts the Guardian process for itself.

You must start **ham** with its full path or with the ***PATH*** variable set to include the path to **ham** as a component.

You must be `root` in order to start or stop a HAM.

Stopping a HAM

To stop the HAM, you must use either the *ham_stop()* (p. 159) function or the **hamctrl** utility. These are the only correct (and the only guaranteed) ways to stop the HAM.

The *ham_stop()* function or the **hamctrl** utility instructs a HAM to terminate. The HAM in turn first instructs the Guardian to terminate, and then terminates itself. To stop the HAM from the command line, use the **hamctrl** utility:

```
hamctrl -stop
```

To stop a remote HAM, use the `-node` option to the `hamctrl` utility:

```
hamctrl -node "nodename" -stop
```

To stop the HAM programmatically using the API, use the following functions:

```
/* terminate                     */
int ham_stop(void);
int ham_stop_nd(int nd);
int ham_stop_node(const char *nodename);
```

Control functions

The following set of functions have been provided to permit control of entities, conditions, and actions that are currently configured.

```
/* control operations                             */
int ham_entity_control(ham_entity_t *ehdl, int command, unsigned flags);
int ham_condition_control(ham_condition_t *chdl, int command, unsigned flags);
int ham_action_control(ham_action_t *ahdl, int command, unsigned flags);
```

The permitted operations (commands) are:

```
HENABLE                 /* enable item           */
HDISABLE                /* disable item          */
HADDFLAGS               /* add flag              */
HREMOVEFLAGS            /* remove flag           */
HSETFLAGS               /* set flag to specific */
HGETFLAGS               /* get flag              */
```

The “enable” and “disable” commands can be used to temporarily unhide/hide an entity, condition, or action.

An entity that is hidden is not removed, but will not be monitored for any conditions. Similarly, a condition that is hidden will never be triggered, while actions that are hidden will not be executed. By default the enable and disable operations do not operate recursively (although the disabling of an entity, will prevent the triggering of any conditions below it, and the disabling of a condition will prevent the execution of the actions in it).

To understand the finer distinctions of the recursive operation of the the control functions refer to the API descriptions for:

- *ham_entity_control()* (p. 150)
- *ham_condition_control()* (p. 121)
- *ham_action_control()* (p. 69)

The “addflags”, “removeflags”, “setflags”, and “getflags” commands can be used to obtain or modify the flags associated with any of the entities, conditions, or actions. For more details, refer to the API descriptions of the *ham_*_control_*()* functions.

Verbosity control

You can use the *ham_verbose()* (p. 161) function to programmatically get or set (increase or decrease) the verbosity:

```
int ham_verbose(const char *nodename, int op, int value);
```

You can also use the `hamctrl` utility to interactively control the verbosity:

```
hamctrl -verbose /* increase    verbosity */
hamctrl +verbose /* decrease    verbosity */
hamctrl =verbose /* get current verbosity */
```

To operate on a remote HAM, use the `hamctrl` utility with the `-node` option:

```
hamctrl -node "nodename" -verbose /* increase    verbosity */
hamctrl -node "nodename" +verbose /* decrease    verbosity */
hamctrl -node "nodename" =verbose /* get current verbosity */
```

where *nodename* is a valid name that represents a remote (or local) node.

Publishing autonomously detected conditions

Entities or other components on the system can publish conditions that they deem interesting to a HAM, and the HAM can in turn deliver these to other components in the system that have expressed interest and subscribed to them. This allows arbitrary system components that are capable of detecting error conditions or potentially erroneous conditions, to report these to the HAM, which in turn can notify other components to start corrective procedures and/or take preventive action.

There are currently two different ways of publishing information to a HAM. Both of these are designed to be general enough to permit clients to build more complex information exchange mechanisms using them.

Publish state transitions

An entity can report its state transitions to a HAM. The HAM maintains the current state of every entity (as reported by the entity). The HAM does not interpret the meaning of the state value itself, neither does it try to validate the state transitions, but can generate events based on transitions from one state to another.

Components can publish transitions that they want the external world to know. These states need not necessarily represent a *specific* state the application uses internally for decision making.

The following function can be used to notify a HAM of a state transition. Since the HAM is only interested in the *next* state in the transition, this is the only information that is transmitted to the HAM. The HAM then triggers a condition state change event internally, which other components can subscribe to, using the *ham_condition_state()* API call described below.

```
/* report a state transition */
int ham_entity_condition_state(ham_entity_t *ehdl, unsigned tostate,
    unsigned flags);
```

Publish other conditions

In addition to the above, components on the system can also publish autonomously detected conditions by using the *ham_entity_condition_raise()* (p. 146) API call. The component raising the condition can also specify a type, class, and severity of its choice, to allow subscribers further granularity in filtering out specific conditions to subscribe to. This call results in the HAM triggering a condition-raise event internally, which other components can subscribe to using the *ham_condition_raise()* API call described below.

```
/* publish autonomously detected condition */
int ham_entity_condition_raise(ham_entity_t *ehdl, unsigned rtype,
    unsigned rclass, unsigned severity, unsigned flags);
```

Subscribing to autonomously published conditions

Subscribers can express their interest in events published by other components by using the following API calls:

- *ham_condition_state()* (p. 131)
- *ham_condition_raise()* (p. 127)

These calls are similar to the *ham_condition()* API call, and return a handle to a condition, but allow the subscriber customize which of several possible published conditions they are interested in.

Trigger based on state transitions

When an entity publishes a state transition, a state transition condition is raised for that entity, based on the two states involved in the transition (the *from* state and the *to* state). Subscribers indicate which states they are interested in by specifying values for the *fromstate* and *tostate* parameters in the API call.

For more details, refer to the API reference documentation for *ham_condition_state()* (p. 131).

```
ham_condition_t *ham_condition_state(ham_entity_t *ehdl, const char *cname,
                 unsigned fromstate, unsigned tostate, unsigned flags);
```

Trigger based on specific published condition

Subscribers can express interest in conditions raised by entities by using *ham_condition_raise()*, indicating as parameters to the call what sort of conditions they are interested in.

For more information, refer to the API documentation for *ham_condition_raise()* (p. 127).

```
ham_condition_t *ham_condition_raise(ham_entity_t *ehdl, const char *cname,
                 unsigned rtype, unsigned rclass, unsigned rseverity,
                 unsigned flags);
```

Chapter 5
Using the Client Recovery Library

The client recovery library provides a drop-in enhancement solution for many standard `libc` I/O operations. The HA library's cover functions provide automatic recovery mechanisms for failed connections that can be recovered from in an HA scenario.

The goal is to provide an API for high availability I/O that can transparently provide recovery to clients, especially in an environment where the servers must also be highly available. The recovery is configurable to tailor specific client needs; we provide examples of ways to develop more complicated recovery mechanisms.

The main principle of the HA library is to provide drop-in replacements for all the "transmission" functions (e.g., *MsgSend*()*). The API lets a client choose specific connections that it would like to make *highly available* — all other connections will operate as ordinary connections.

Normally, when a server that the client is talking to fails, or if there's a transient network fault, the *MsgSend*()* functions return an error indicating that the connection ID (or file descriptor) is stale or invalid (EBADF).

In an HA-aware scenario, these transient faults are often recovered from almost immediately (on the server end), thus making the services available again. Unfortunately, clients using a standard I/O offering might not be available to benefit from this to the maximum unless they provide mechanisms to recover from these errors, and then retransmit the information/data, which often might involve a nontrivial rework of client programs.

By providing/achieving recovery inside the HA library itself, we can automatically take advantage of the HA-aware services that restart themselves or are automatically restarted or of the services that are provided in a transparent cluster/redundant way.

Since recovery itself is a connection-specific task, we allow clients to provide recovery mechanisms that will be used to restore connections when they fail. Irrecoverable errors are propagated back reliably so that any client that doesn't wish to recover will get the I/O library semantics that it expects.

The recovery mechanism can be anything ranging from a simple reopen of the connection to a more complex scenario that includes the retransmission/renegotiation of connection-specific information.

MsgSend()* functions

Normally, the *MsgSend*()* functions return EBADF/ESRCH when a connection is stale or closed on the server end (e.g., because the server dies). In many cases, the servers themselves return (e.g., they're restarted) and begin to offer the services properly almost immediately (in an HA scenario). Rather than merely terminate the message transmission with an error, in some cases it might be possible to perform recovery and continue with the message transmission.

The HA library functions that "cover" all the *MsgSend*()* varieties are designed to do exactly this. When a specific invocation of one of the *MsgSend*()* functions fails, a client-provided recovery function is called. This recovery function can attempt to reestablish the connection and return control to the HA library's *MsgSend*()* function. As long as the connection ID returned by the recovery function is the same as the old connection ID (which in many cases is easy to ensure via *close/open/dup2()* sequences), then the *MsgSend*()* functions can now attempt to retransmit the data.

If at any point the errors returned by *MsgSend*()* are anything other than EBADF/ESRCH, these errors are propagated back to the client. Note also that if the connection ID isn't an HA-aware connection ID, or if the client hasn't provided a recovery function or that function can't re-obtain the same connection ID, then the error is allowed to propagate back to the client to handle in whatever way it likes.

Clients can change their recovery functions. And since clients can also pass around "recovery/connection" information (which in turn is passed by the HA library to the recovery function), clients can construct complex recovery mechanisms that can be modified dynamically.

The client-side recovery library lets clients reconstruct the state required to continue the message transmission after reconnecting to either the same server or to a different server. The client is responsible for determining what constitutes the state that must be reconstructed and for performing this appropriately while the recovery function is called.

Other covers and convenience functions

In addition to the cover functions for the standard *MsgSend*()* calls, the HA library provides clients with two “HA-awareness” functions that let you designate a connection as being HA-aware or similarly remove such a designation for an already HA-aware connection:

HA-awareness functions

ha_attach() **(p. 164)**

Associate a recovery function with a connection to make it HA-aware.

ha_detach()

Remove a previously specified association between a recovery function and a connection. This makes the connection no longer HA-aware.

ha_connection_ctrl() **(p. 168)**

Control the operation of a HA-aware connection.

I/O covers

The HA library also provides the following cover functions whose behavior is essentially the same as the original functions being covered, but augmented slightly where the connections are also HA-aware:

ha_open(), ha_open64() **(p. 183)**

Open a connection and attach it to the HA lib. These functions, in addition to calling the underlying *open* calls also make the connections HA-aware by calling *ha_attach()* automatically. As a result, using these calls is equivalent to calling *open()* or *open64()* and following that with a call to *ha_attach()*.

ha_creat(), ha_creat64() **(p. 174)**

Create a connection and attach it to the HA lib. These functions, in addition to calling the underlying *creat* calls also make the connections HA-aware by calling *ha_attach()* automatically. As a result, using these calls is equivalent to calling *creat()* or *creat64()* and following that with a call to *ha_attach()*.

ha_ConnectAttach(), ha_ConnectAttach_r() **(p. 170)**

Create a connection using *ConnectAttach()* and attach it to the HA lib. These functions, in addition to calling the underlying *ConnectAttach* calls also make the connections HA-aware by calling *ha_attach()* automatically. As a result, using these calls is equivalent to calling *ConnectAttach()* or *ConnectAttach_r()* and following that with a call to *ha_attach()*.

ha_ConnectDetach(), ha_ConnectDetach_r() **(p. 172)**

Detach an attached *fd*, then close the connection using *ConnectDetach()*. These functions, in addition to calling the underlying *ConnectDetach* calls also make the connections HA-aware by calling *ha_attach()* automatically. As a result, using these calls is equivalent to calling *ConnectDetach()* or *ConnectDetach_r()* and following that with a call to *ha_attach()*.

ha_fopen() **(p. 181)**

Open a file stream and attach it to the HA lib. This function, in addition to calling the underlying *fopen()* call also makes connections HA-aware by calling *ha_attach()* automatically. As a result, using this call is equivalent to calling *fopen()* and following that with a call to *ha_attach()*.

ha_fclose() **(p. 180)**

Detach an attached HA *fd* for a file stream, then close it. This function, in addition to calling the underlying *fclose()* call also makes connections HA-aware by calling *ha_attach()* automatically. As a result, using this call is equivalent to calling *fclose()* and following that with a call to *ha_attach()*.

ha_close() **(p. 167)**

Detach an attached HA *fd*, then close it. This function, in addition to calling the underlying *close()* call also makes connections HA-aware by calling *ha_attach()* automatically. As a result, using this call is equivalent to calling *close()* and following that with a call to *ha_attach()*.

ha_dup() **(p. 178)**

Duplicate an HA connection. This function, in addition to calling the underlying *dup()* call also makes connections HA-aware by calling *ha_attach()* automatically. As a result, using this call is equivalent to calling *dup()* and following that with a call to *ha_attach()*.

Convenience functions

In addition to the covers, the library also provides these two convenience functions that reopen connections for recovery:

ha_reopen() **(p. 187)**

Reopen a connection while performing recovery.

ha_ReConnectAttach() **(p. 185)**

Reopen a connection while performing recovery.

For descriptions of all of the HA library functions, see the *Client Recovery Library Reference* (p. 163) chapter in this guide.

A simple example

Here's a simple example of a client that has a connection open to a server and tries to read data from it. After reading from the descriptor, the client goes off to do something else (possibly causing a delay), and then returns to read again.

During this window of delay, the server might have died and returned, in which case the initial connection to the server (that has died) is now stale.

But since the connection has been made HA-aware, and a recovery function has been associated with it, the connection is able to reestablish itself.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <ha/cover.h>

#define SERVER "/path/to/server"

typedef struct handle {
    int nr;
} Handle ;

int recover_conn2(int oldfd, void *hdl)
{
    int newfd;
    Handle *thdl;
    thdl = (Handle *)hdl;
    printf("recovering for fd %d  inside function 2\n",oldfd);
    /* re-open the connection */
    newfd = ha_reopen(oldfd, SERVER, O_RDONLY);
  /* perform any other kind of state re-construction */
    (thdl->nr)++;
    return(newfd);
}

int recover_conn(int oldfd, void *hdl)
{
    int newfd;
    Handle *thdl;
    thdl = (Handle *)hdl;
    printf("recovering for fd %d inside function\n",oldfd);
    /* re-open the connection */
    newfd = ha_reopen(oldfd, SERVER, O_RDONLY);
    /* perform any other kind of state reconstruction */
    (thdl->nr)++;
    return(newfd);
}

int main(int argc, char *argv[])
{
    int status;
    int fd;
    int fd2;
    int fd3;
    Handle hdl;
    char buf[80];
    int i;

    hdl.nr = 0;
    /* open a connection and make it HA aware */
    fd = ha_open(SERVER, O_RDONLY,recover_conn, (void *)&hdl, 0);
    if (fd < 0) {
        printf("could not open %s\n", SERVER);
        exit(-1);
    }

    printf("fd = %d\n",fd);
  /* Dup the FD. the copy will also be HA aware */
    fd2 = ha_dup(fd);

    printf("dup-ped fd2 = %d\n",fd2);
    printf("before sleeping first time\n");

  /*
   Go to sleep...
   Possibly the SERVER might die and return in this little
   time period.
  */
    sleep(15);

  /*
   reading from dup-ped fd
   this should work just normally if SERVER has not died.
   But if the SERVER has died and returned, the
   initial read will fail, but the recovery function
   will be called, and it will re-establish the
   connection, and then re-establish the current
   file position and then re-issue the read call
   which should succeed now.
  */

    printf("trying to read from %s using fd %d\n",SERVER, fd2);
    status = read(fd2,buf,30);
    if (status < 0)
        printf("error: %s\n",strerror(errno));

  /*
   fd and fd2 are dup-ped fd's
   changing the recovery function for fd2
   From this point forwards, the recovery (if at all)
   will performed using "recover_conn2" as the recovery
   function.
  */

    status = ha_attach(fd2, recover_conn2, (void *)&hdl, HAREPLACERECOVERYFN);

    ha_close(fd); /* close fd */

  /* open a new connection */
    fd = open(SERVER, O_RDONLY);
    printf("New fd = %d\n",fd);

  /* make it HA aware. */
    status = ha_attach(fd, recover_conn, (void *)&hdl, 0);

    printf("before sleeping again\n");

  /* copy it again */
    fd3 = ha_dup(fd);

  /* go to sleep...possibly another option for the server to fail. */
    sleep(15);

  /*
   get rid of one of the fd's
   we still have a copy in fd3, which must have the
   recovery functions associated with it.
  */
    ha_close(fd);

    printf("trying to read from %s using fd %d\n",SERVER, fd3);

  /*
   if it fails, the call will generate a call back to the
   recovery function "recover_conn"
  */
    status = read(fd3,buf,30);
    if (status < 0)
        printf("error: %s\n",strerror(errno));

    printf("trying to read from %s once more using fd %d\n",SERVER, fd2);
  /*
   if this call fails, recovery will be via the
   second function "recover_conn2", since we replaced
   the function for fd2.
  */
    status = read(fd2,buf,30);
    if (status < 0)
        printf("error: %s\n",strerror(errno));

  /* close the fd2, and detach it from the HA lib */
    ha_close(fd2);

  /*
   finally print out our local statistics that we have been
   retaining along the way.
  */
    printf("total recoveries, %d\n",hdl.nr);
    exit(0);
}
```

State-reconstruction example

In the following example, in addition to reopening the connection to the server, the client also reconstructs the state of the connection by seeking to the current file (connection) offset.

This example also shows how the client can maintain state information that can be used by the recovery functions to return to a previously check-pointed state before the failure, so that the message transmission can continue properly.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <ha/cover.h>

#define REMOTEFILE "/path/to/remote/file"

typedef struct handle {
    int nr;
    int curr_offset;
} Handle ;

int recover_conn(int oldfd, void *hdl)
{
    int newfd;
    int newfd2;
    Handle *thdl;
    thdl = (Handle *)hdl;
    printf("recovering for fd %d inside function\n",oldfd);
   /* re-open the file */
    newfd = ha_reopen(oldfd, REMOTEFILE , O_RDONLY);
   /* re-construct state, by seeking to the correct offset. */
    if (newfd >= 0)
      lseek(newfd, thdl->curr_offset, SEEK_SET);
    (thdl->nr)++;
    return(newfd);
}

int main(int argc, char *argv[])
{
    int status;
    int fd;
    int fd2;
    int fd3;
    Handle hdl;
    char buf[80];
    int i;

    hdl.nr = 0;
    hdl.curr_offset = 0;
    /* open a connection */
    fd = ha_open(REMOTEFILE, O_RDONLY,recover_conn,
                 (void *)&hdl, 0);
    if (fd < 0) {
        printf("could not open file\n");
        exit(-1);
    }
    fd2 = open(REMOTEFILE, O_RDONLY);
    printf("trying to read from file using fd %d\n",fd);
    printf("before sleeping first time\n");
    status = read(fd,buf,15);
    if (status < 0)
        printf("error: %s\n",strerror(errno));
    else {
        for (i=0; i < status; i++)
            printf("%c",buf[i]);
        printf("\n");
   /*
    update state of the connection
    this is a kind of checkpointing method.
    we remember state, so that the recovery functions
    have an easier time.
   */
        hdl.curr_offset += status;
    }

    fd3 = ha_dup(fd);
    sleep(18);
   /*
    sleep for some arbitrary period
    this could be some other computation
    or some other blocking operation, which gives
    a window within which the server might fail
   */

   /* reading from dup-ped fd */
    printf("trying to read from file using fd %d\n",fd);
    printf("after sleeping\n");

   /*
    if the read initially fails
    it will recover, re-open and seek to the right spot!!
   */
    status = read(fd,buf,15);
    if (status < 0)
        printf("error: %s\n",strerror(errno));
    else {
        for (i=0; i < status; i++)
            printf("%c",buf[i]);
        printf("\n");
        hdl.curr_offset += status;
    }
    printf("trying to read from file using fd %d\n",fd2);
   /*
    try it again.. this time using the copy.
    recovery will again happen upon failure,
    automatically re-connecting/seeking etc.
   */
    status = read(fd2,buf,15);
    if (status < 0)
        printf("error: %s\n",strerror(errno));
    else {
        for (i=0; i < status; i++)
            printf("%c",buf[i]);
        printf("\n");
    }
    printf("total recoveries, %d\n",hdl.nr);
    ha_close(fd);
    close(fd2);
    exit(0);
}
```

Chapter 6
HAM API Reference

The High Availability Framework includes the following functions you can use in your applications to interact with a HAM:

Function	Description
ham_action_control() (p. 69)	Perform control operations on an action object in a HAM.
ham_action_execute() (p. 72)	Add an execute action to a condition.
ham_action_fail_execute() (p. 75)	Add an execute action to an action, that will be executed if the corresponding action fails.
ham_action_fail_log() (p. 78)	Insert a log message into the activity log of a HAM.
ham_action_fail_notify_pulse() (p. 80)	Add a notify pulse action to an action, that will be executed if the corresponding action fails.
ham_action_fail_notify_pulse_node() (p. 80)	Add a notify pulse action to an action, that will be executed if the corresponding action fails, using a nodename.
ham_action_fail_notify_signal() (p. 83)	Add a notify signal action to an action, that will be executed if the corresponding action fails.
ham_action_fail_notify_signal_node() (p. 83)	Add a notify signal action to an action, that will be executed if the corresponding action fails, using a nodename.
ham_action_fail_waitfor() (p. 86)	Add a waitfor action to an action, that will be executed if the corresponding action fails
ham_action_handle() (p. 88)	Get a handle to an action in a condition in an entity.
ham_action_handle_node() (p. 88)	Get a handle to an action in a condition in an entity, using a nodename.

Function	Description
ham_action_handle_free() (p. 90)	Free a previously obtained handle to an action in a condition in an entity.
ham_action_heartbeat_healthy() (p. 92)	Reset a heartbeat's state to healthy.
ham_action_log() (p. 94)	Insert a log message into the activity log of the HAM.
ham_action_notify_pulse() (p. 96)	Add a notify-pulse action to a condition.
ham_action_notify_pulse_node() (p. 96)	Add a notify-pulse action to a condition, using a nodename.
ham_action_notify_signal() (p. 99)	Add a notify-signal action to a condition.
ham_action_notify_signal_node() (p. 99)	Add a notify-signal action to a condition, using a nodename.
ham_action_remove() (p. 102)	Remove an action from a condition.
ham_action_restart() (p. 104)	Add a restart action to a condition.
ham_action_waitfor() (p. 107)	Add a waitfor action to a condition.
ham_attach() (p. 110)	Attach an entity.
ham_attach_node() (p. 110)	Attach an entity, using a nodename.
ham_attach_self() (p. 114)	Attach an application as a self-attached entity.
ham_condition() (p. 117)	Set up a condition to be triggered when a certain event occurs.
ham_condition_control() (p. 121)	Perform control operations on a condition object in a HAM.
ham_condition_handle() (p. 123)	Get a handle to a condition in an entity.
ham_condition_handle_node() (p. 123)	Get a handle to a condition in an entity, using a nodename.
ham_condition_handle_free() (p. 125)	Free a previously obtained handle to a condition in an entity.
ham_condition_raise() (p. 127)	Attach a condition associated with a condition raise condition that is triggered by an entity raising a condition.
ham_condition_remove() (p. 129)	Remove a condition from an entity.

Function	Description
ham_condition_state() (p. 131)	Attach a condition associated with a state transition condition that is triggered by an entity reporting a state change.
ham_connect() (p. 134)	Connect to a HAM.
ham_connect_nd() (p. 134)	Connect to a remote HAM.
ham_connect_node() (p. 134)	Connect to a remote HAM, using a nodename.
ham_detach() (p. 136)	Detach an entity from a HAM.
ham_detach_name() (p. 138)	Detach an entity from a HAM, using an entity name.
ham_detach_name_node() (p. 138)	Detach an entity from a HAM, using an entity name and a nodename.
ham_detach_self() (p. 140)	Detach a self-attached entity from a HAM.
ham_disconnect() (p. 142)	Disconnect from a HAM.
ham_disconnect_nd() (p. 142)	Disconnect from a remote HAM.
ham_disconnect_node() (p. 142)	Disconnect from a remote HAM, using a nodename.
ham_entity() (p. 144)	Create entity placeholder objects in a HAM.
ham_entity_condition_raise() (p. 146)	Used by an entity to raise a condition.
ham_entity_condition_state() (p. 148)	Used by an entity to notify the HAM of a state transition.
ham_entity_control() (p. 150)	Perform control operations on an entity object in a HAM.
ham_entity_handle() (p. 153)	Get a handle to an entity.
ham_entity_handle_node() (p. 153)	Get a handle to an entity, using a nodename.
ham_entity_handle_free() (p. 155)	Free a previously obtained handle to an entity.
ham_entity_node() (p. 144)	Create entity placeholder objects in a HAM, using a nodename.
ham_heartbeat() (p. 157)	Send a heartbeat to a HAM.

Function	Description
ham_stop() (p. 159)	Stop a HAM.
ham_stop_nd() (p. 159)	Stop a remote HAM.
ham_stop_node() (p. 159)	Stop a remote HAM, using a nodename.
ham_verbose() (p. 161)	Modifies the verbosity of a HAM.

ham_action_control()

Perform control operations on an action object in a HAM

Synopsis:

```
#include <ha/ham.h>

int ham_action_control( ham_action_t *ahdl,
                        int command,
                        unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_control()* function can be used to control the state of an action object in a HAM. This function is designed to be extensible with additional commands. Currently, the following commands are defined:

`HENABLE`

Enable the action.

`HDISABLE`

Disable the action.

`HADDFLAGS`

Add the flags.

`HREMOVEFLAGS`

Remove the flags.

`HSETFLAGS`

Set the flags to the given value.

`HGETFLAGS`

Get the current flags.

When an action item is enabled (the default), it's executed when the condition associated with it is triggered. When an action item is disabled, the action *isn't* executed when the condition associated with it is triggered. Individual conditions and entities can be enabled and disabled using the corresponding control functions for conditions and entities, respectively.

The add flags, remove flags, and set flags commands can be used to modify the set of flags associated with the entity being controlled. Add flags and remove flags are used to either add to or remove from the current set of flags, the specified set of flags (as given in *flags*). The set flags function is called when the current set of flags is to be replaced by *flags*.

Flags

Any flag that is valid for the corresponding action can be used when *ham_action_control()* is being used to set flags, with the exception of `HACTIONDONOW`.

For the `HENABLE` and `HDISABLE` commands:

`HRECURSE`

Applies the command recursively.

Returns:

For the enable, disable, add flags, remove flags, and set flags functions:

0

Success.

-1

An error occurred (*errno* is set).

For the get flags function:

flags

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The *command* or *flags* variable is invalid.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_execute()

Add an execute action to a condition

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_execute(
                  ham_condition_t *chdl,
                  const char *aname,
                  const char *path,
                  unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_execute()* function adds an action (*aname*) to the specified condition. The action will execute an external program or command specified by *path*. The *path* parameter must contain the FULL path to the executable along with all parameters to be passed to it. If either the pathname or the arguments contain spaces that need to be passed on literally to the spawn call, they need to be quoted. As long as the subcomponents within the *path* arguments are quoted, using either of the following methods:

```
\'path with space\'
```

or

```
\"path with space\",
```

the following is allowed:

```
"\'path with space\' arg1 arg2 \"arg3 with space\"".
```

This would be parsed as

```
"path with space" -> path

arg1 = arg1

arg2 = arg2

arg3 = "arg3 with space".
```

The command line is in turn passed onto a *spawn* command by the HAM to create a new process that will execute the command.

The handle (*chdl*) is obtained either:

- from one of the *ham_condition*()* functions to add conditions

  or:

- by calling any of the *ham_condition_handle()* functions to request a handle to a specific condition.

The action is executed when the appropriate condition is triggered.

Currently the following flags are defined:

HACTIONDONOW

Tells the HAM to perform the action once immediately, in addition to performing it whenever the condition is triggered.

HREARMAFTERRESTART

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

HACTIONBREAKONFAIL

Indicates that if this action were to fail, and it is part of a list of actions, none of the actions following this one in the list of actions will be executed.

HACTIONKEEPONFAIL

Indicates that the action will be retained even if it fails. The default behavior is to remove failed actions. Nevertheless if the condition that this action is associated with is not retained, the action will get automatically removed.

Users can specify what will be done if an action fails. By adding action fail *actions* to a list associated with an action, users can determine what will be done in the case of an action failure. For every action that fails, the corresponding action fail list will be executed. Certain actions (e.g., *ham_action_log()* and *ham_action_heartbeat_healthy()*) never fail. For more details, refer to the appropriate section in the HAM API reference for the *ham_action_fail_*()* calls.

Returns:

A valid handle to an action to a condition, or NULL if an error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_execute()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*chdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_fail_execute()

Add an execute action to an action, that will be executed if the corresponding action fails

Synopsis:

```
#include <ha/ham.h>

int ham_action_fail_execute(
                 ham_action_t *ahdl,
                 const char *aname,
                 const char *path,
                 unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_fail_execute()* function adds an action fail item (*aname*) to the specified action. The action will execute an external program or command specified by *path*. The *path* parameter must contain the FULL path to the executable along with all parameters to be passed to it. along with all parameters to be passed to it. If either the pathname or the arguments contain spaces that need to be passed on literally to the spawn call, they need to be quoted. As long as the subcomponents within the *path* arguments are quoted, using either of the following methods:

```
\'path with space\'
```

or

```
\"path with space\",
```

the following is allowed:

```
"\'path with space\' arg1 arg2 \"arg3 with space\"".
```

This would be parsed as

```
"path with space" -> path

arg1 = arg1

arg2 = arg2

arg3 = "arg3 with space".
```

The command line is in turn passed onto a *spawn()* command by the HAM to create a new process that will execute the command.

The handle (*ahdl*) is obtained either:

- from one of the *ham_action*()* functions to add actions

  or:

- by calling any of the *ham_action_handle()* functions to request a handle to a specific action.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The name given in *aname* is invalid (e.g., it contains the “`/`” character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_execute()* aren't the same.

ENAMETOOLONG

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

ENOENT

There's no entity or condition specified by the given handle (*ahdl*).

ENOMEM

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_fail_log()

Insert a log message into the activity log of a HAM

Synopsis:

```
#include <ha/ham.h>

int ham_action_fail_log(
                     ham_action_t *ahdl,
                     const char *aname,
                     const char *msg,
                     unsigned attachprefix,
                     int verbosity,
                     unsigned flags);
```

Library:

`libham`

Description:

You can use the *ham_action_fail_log()* function to insert log messages into the activity log stream that a HAM maintains. This action is executed when the corresponding action that it is associated with fails.

The handle (*ahdl*) is obtained either:

- from one of the *ham_action*()* functions to add actions

  or:

- by calling any of the *ham_action_handle()* functions to request a handle to a specific action.

The log message to be inserted is specified by *msg*, and will be generated if the verbosity of the HAM is greater than or equal to the value specified in *verbosity*. Also, if *attachprefix* is non-zero, a prefix will be added to the log message that contains the current entity/condition/action that this message is related to.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_restart()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*ahdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_fail_notify_pulse(), ham_action_fail_notify_pulse_node()

Add a notify pulse action to a an action, that will be executed if the corresponding action fails

Synopsis:

```
#include <ha/ham.h>

int ham_action_fail_notify_pulse(
                 ham_action_t *ahdl,
                 const char *aname,
                 int nd,
                 pid_t topid,
                 int chid,
                 int pulsecode,
                 int value,
                 unsigned flags);

int ham_action_fail_notify_pulse_node(
                 ham_action_t *ahdl,
                 const char *aname,
                 const char *nodename,
                 pid_t topid,
                 int chid,
                 int pulsecode,
                 int value,
                 unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_fail_notify_pulse*()* functions add an action fail item (*aname*) to the specified action. The action will deliver a pulse specified by *pulsecode* to the specified *nd*/*pid*/*chid* or the *nodename*/*pid*/*chid* with value given by *value*. The *nd* specified to *ham_action_notify_pulse()* is the node identifier of the recipient node. This *nd* must be valid at the time the call is made.

The handle (*ahdl*) is obtained either:

- from one of the *ham_action*()* functions to add actions

  or:

- by calling any of the *ham_action_handle()* functions to request a handle to a specific action.

The action is executed when the corresponding action that it is associated with, fails.

Currently, there are no flags defined.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The name given in *aname* is invalid (e.g., it contains the "/" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_execute()* aren't the same.

ENAMETOOLONG

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

ENOENT

There's no entity or condition specified by the given handle (*ahdl*).

ENOMEM

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

ham_action_fail_notify_signal(), ham_action_fail_notify_signal_node()

Add a notify signal action to a an action, that will be executed if the corresponding action fails

Synopsis:

```
#include <ha/ham.h>

int ham_action_fail_notify_signal(
                  ham_action_t *ahdl,
                  const char *aname,
                  int nd,
                  pid_t topid,
                  int signum,
                  int code,
                  int value,
                  unsigned flags);

int ham_action_fail_notify_signal_node(
                  ham_action_t *ahdl,
                  const char *aname,
                  const char *nodename,
                  pid_t topid,
                  int signum,
                  int code,
                  int value,
                  unsigned flags);
```

Library:

```
libham
```

Description:

The *ham_action_fail_notify_signal*()* functions add an action fail item (*aname*) to the specified action. The action will deliver a signal specified by *signum* to the specified *nd*/*pid* or *nodename*/*pid* with code specified in *code* and value given by *value*. The *nd* specified to *ham_action_notify_signal()* is the node identifier of the recipient node. This *nd* must be valid at the time the call is made.

The handle (*ahdl*) is obtained either:

- from one of the *ham_action*()* functions to add actions

  or:

- by calling any of the *ham_action_handle()* functions to request a handle to a specific action.

The action is executed when the corresponding action that it is associated with, fails.

Currently, there are no flags defined.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "/" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_execute()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*ahdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

ham_action_fail_waitfor()

Add a waitfor action to an action, that will be executed if the corresponding action fails

Synopsis:

```
#include <ha/ham.h>

int ham_action_fail_waitfor(
                 ham_action_t *ahdl,
                 const char *aname,
                 const char *path,
                 int delay,
                 unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_fail_waitfor()* function adds an action fail item (*aname*) to the specified action. The action will either delay for *delay* milliseconds or wait until *path* (if specified) appears in the name space (whichever is earlier). The *path* parameter must contain the FULL path that is being *watched* for.

The handle (*ahdl*) is obtained either:

- from one of the *ham_action*()* functions to add actions

  or:

- by calling any of the *ham_action_handle()* functions to request a handle to a specific action.

The action is executed when the corresponding action that it is associated with, fails.

Currently, there are no flags defined.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_execute()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*ahdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_handle(), ham_action_handle_node()

Get a handle to an action in a condition in an entity

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_handle(int nd,
                                const char *ename,
                                const char *cname,
                                const char *aname,
                                unsigned flags);

ham_action_t *ham_action_handle_node(int nd,
                                const char *nodename,
                                const char *ename,
                                const char *cname,
                                const char *aname,
                                unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_handle()* function returns a handle to an action (*aname*) in a condition (*cname*) in an entity (*ename*).

You can pass the handle obtained from this function to other functions that expect a handle (e.g., *ham_action_remove()* (p. 102) or *ham_action_handle_free()* (p. 90)).

To get a handle for a global entity, pass `NULL` for *ename*.

The handle returned is opaque — its contents are internal to the library.

If a node (*nd*) is specified, the handle is to an entity/condition/action combination that refers to a process on that remote node. The *ham_action_handle_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*).

There are no flags defined at this time.

Returns:

A valid `ham_action_t`, or `NULL` if an error occurred (*errno* is set).

Errors:

`EINVAL`

The name given in *ename*, *cname*, or *aname* is invalid (e.g., it contains the "`/`" character), or *cname* or *aname* is `NULL`.

`ENAMETOOLONG`

The name given is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no action of the name *aname* in a condition *cname* defined in an entity *ename* in the HAM's current context.

`ENOMEM`

Not enough memory to create a new handle.

Classification:

QNX Neutrino

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

Caveats:

A call to *ham_action_handle()* and a subsequent use of the handle returned in a call such as *ham_action_remove()* are completely *asynchronous*. Therefore, a valid action/condition/entity may no longer exist when the handle is used at a later time.

In such an event, the *ham_action*()* functions will return an error (`ENOENT`) that the action in the condition doesn't exist in the given entity.

ham_action_handle_free()

Free a previously obtained handle to an action in a condition in an entity

Synopsis:

```
#include <ha/ham.h>

int ham_action_handle_free(ham_action_t *ahdl);
```

Library:

`libham`

Description:

The *ham_action_handle_free()* function frees a handle (*ahdl*) associated with a given action in a condition in an entity. The function reclaims all storage associated with the handle.

The handle you pass as an argument (*ahdl*) must be obtained from *ham_action_execute()* (p. 72), *ham_action_restart()* (p. 104), *ham_action_notify_pulse()* (p. 96), *ham_action_notify_signal()* (p. 99), or *ham_action_waitfor()* (p. 107).

Once a handle is freed, it is no longer available to refer to any action. The *ham_action_handle_free()* call frees storage allocated for the handle, but does not remove the action itself, which is in the HAM.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EINVAL`

The handle isn't valid.

Classification:

QNX Neutrino

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

Caveats:

The *ham_action_handle_free()* function frees storage related only to the handle in the client — it doesn't remove the corresponding action.

ham_action_heartbeat_healthy()

Reset a heartbeat's state to healthy

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_heartbeat_healthy(
                  ham_condition_t *chdl,
                  const char *aname,
                  unsigned flags)
```

Description:

You use this function to reset the state of a heartbeat to healthy so that HAM can resume monitoring. Assuming that the client missed one or more heartbeats (i.e., the condition `CONDHBEATMISSEDLOW|HIGH` is true), and that a recovery has been performed, the *ham_action_heartbeat_healthy()* call instructs HAM to monitor the client again.

The following flag is currently defined:

HREARMAFTERRESTART

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

Returns:

A valid handle to an action to a condition, or `NULL` if an error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EEXIST

This action already exists in the specified condition.

EINVAL

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_log()

Insert a log message into the activity log of the HAM

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_log(
                    ham_condition_t *chdl,
                    const char *aname,
                    const char *msg,
                    unsigned attachprefix,
                    int verbosity,
                    unsigned flags);
```

Library:

`libham`

Description:

You can use the *ham_action_log()* function to insert log messages into the activity log stream that the HAM maintains.

The handle (*chdl*) is obtained either:

- from one of the *ham_condition*()* functions to add conditions

  or:

- by calling any of the *ham_condition_handle()* functions to request a handle to a specific condition.

The log message to be inserted is specified by *msg*, and will be generated if the verbosity of the HAM is greater than or equal to the value specified in *verbosity*. Also, if *attachprefix* is non-zero, a prefix will be added to the log message that contains the current *entity/condition/action* that this message is related to.

The following flag is currently defined:

HREARMAFTERRESTART

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

Returns:

A valid handle to an action to a condition, or NULL if an error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_restart()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*chdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_notify_pulse(), *ham_action_notify_pulse_node()*

Add a notify-pulse action to a condition

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_notify_pulse(
                     ham_condition_t *chdl,
                     const char *aname,
                     int nd,
                     pid_t topid,
                     int chid,
                     int pulsecode,
                     int value,
                     unsigned flags);

ham_action_t *ham_action_notify_pulse_node(
                     ham_condition_t *chdl,
                     const char *aname,
                     const char *nodename,
                     pid_t topid,
                     int chid,
                     int pulsecode,
                     int value,
                     unsigned flags);
```

Library:

```
libham
```

Description:

These functions add an action to a given condition. This action will deliver a pulse notification to the process given in *topid*.

The handle (*chdl*) is obtained either:

- from one of the *ham_condition*()* functions to add conditions

  or:

- by calling any of the *ham_condition_handle()* functions to request a handle to a specific condition.

The action is executed when the appropriate condition is triggered.

The *nd* specifies the node identifier of the remote node (or local node) to which the pulse is targeted The *nd* specified is the node identifier of the remote node at the time the call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

Use the *ham_action_notify_pulse_node()* function when a nodename is used to specify a remote HAM instead of a node identifier (*nd*). The *ham_action_notify_pulse_node()* function takes as a parameter a fully qualified node name (FQNN)

The pulse in *pulsecode* with the given *value* will be delivered to *topid* on the specified channel ID (*chid*).

Currently the following flag is defined:

HREARMAFTERRESTART

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

HACTIONBREAKONFAIL

Indicates that if this action were to fail, and it is part of a list of actions, none of the actions following this one in the list of actions will be executed.

HACTIONKEEPONFAIL

Indicates that the action will be retained even if it fails. The default behavior is to remove failed actions. Nevertheless if the condition that this action is associated with is not retained, the action will get automatically removed.

Users can specify what will be done if an action fails. By adding action fail *actions* to a list associated with an action, users can determine what will be done in the case of an action failure. For every action that fails, the corresponding action fail list will be executed. Certain actions (e.g., *ham_action_log()* and *ham_action_heartbeat_healthy()*) never fail. For more details, refer to the appropriate section in the HAM API reference for the *ham_action_fail_*()* calls.

Returns:

A valid `ham_action_t` or NULL if an error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EEXIST

The name provided for the action already exists.

EINVAL

The name given in *aname* is invalid (e.g., it contains the "/" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_notify_pulse()* aren't the same.

ENAMETOOLONG

The name given is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

ENOENT

There's no action of the name *aname* in a condition *cname* defined in an entity *ename* in the HAM's current context.

ENOMEM

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_notify_signal(), ham_action_notify_signal_node()

Add a notify-signal action to a condition

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_notify_signal(
                 ham_condition_t *chdl,
                 const char *aname,
                 int nd,
                 pid_t topid,
                 int signum,
                 int code,
                 int value,
                 unsigned flags);

ham_action_t *ham_action_notify_signal_node(
                 ham_condition_t *chdl,
                 const char *aname,
                 const char *nodename,
                 pid_t topid,
                 int signum,
                 int code,
                 int value,
                 unsigned flags);
```

Library:

```
libham
```

Description:

These functions add an action to a given condition. The action will deliver a signal notification to the process given in *topid*.

The handle (*chdl*) is obtained either:

- from one of the *ham_condition*()* functions to add conditions

  or:

- by calling any of the *ham_condition_handle()* functions to request a handle to a specific condition.

The action is executed when the appropriate condition is triggered.

You use the node descriptor (*nd*) to specify where in the network the recipient of the notification resides.

The *nd* specifies the node identifier of the remote (or local) node to which the signal is targeted The *nd* specified is the node identifier of the remote node at the time the call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

Use the *ham_action_notify_signal_node()* function when a nodename is used to specify a remote HAM instead of a node identifier (*nd*). The *ham_action_notify_signal_node()* function takes as a parameter a fully qualified node name (FQNN)

The signal in *signum* with the given *value* will be delivered to the process *pid* on node *nd*. This can also be used to terminate processes by sending them appropriate signals like `SIGTERM`, `SIGKILL` etc.

Currently the following flags are defined:

`HREARMAFTERRESTART`

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

`HACTIONBREAKONFAIL`

Indicates that if this action were to fail, and it is part of a list of actions, none of the actions following this one in the list of actions will be executed.

`HACTIONKEEPONFAIL`

Indicates that the action will be retained even if it fails. The default behavior is to remove failed actions. Nevertheless if the condition that this action is associated with is not retained, the action will get automatically removed.

Users can specify what will be done if an action fails. By adding action fail *actions* to a list associated with an action, users can determine what will be done in the case of an action failure. For every action that fails, the corresponding action fail list will be executed. Certain actions (e.g., *ham_action_log()* and *ham_action_heartbeat_healthy()*) never fail. For more details, refer to the appropriate section in the HAM API reference for the *ham_action_fail_*()* calls.

Returns:

A valid handle to an action to a condition, or `NULL` if an error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EEXIST

The name provided for the action already exists.

EINVAL

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_notify_signal()* aren't the same.

ENAMETOOLONG

The name given is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

ENOENT

There's no action of the name *aname* in a condition *cname* defined in an entity *ename* in the HAM's current context.

ENOMEM

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_remove()

Remove an action from a condition

Synopsis:

```
#include <ha/ham.h>

int ham_action_remove( ham_action_t *ahdl,
                       unsigned flags);
```

Library:

`libham`

Description:

You use the *ham_action_remove()* function to remove an action from a condition in a specific entity.

You can obtain the handle (*ahdl*) either:

- from one of the *ham_action*()* functions that add actions

  or:

- by calling *ham_action_handle()* (p. 88) to request a handle to a specific condition.

The *flags* argument isn't currently used.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The handle passed is invalid.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_remove()* aren't the same.

ENOENT

There's no action corresponding to the given handle (*ahdl*).

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_restart()

Add a restart action to a condition

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_restart( ham_condition_t *chdl,
                                  const char *aname,
                                  const char *path,
                                  unsigned flags);
```

Library:

`libham`

Description:

You use the *ham_action_restart()* function to add an action (*aname*) to a given condition. In this case, the action will restart a program that has died.

Restart actions can be associated only with death conditions (`CONDDEATH`).

Note also that there can be only one restart action over all the death conditions in an entity.

The handle (*chdl*) is obtained either:

- from one of the *ham_condition*()* functions to add conditions

  or:

- by calling any of the *ham_condition_handle()* functions to request a handle to a specific condition.

You use the *path* parameter to specify the external program or command line to be executed — *path* must contain the FULL path to the executable along with all parameters to be passed to it. along with all parameters to be passed to it. If either the pathname or the arguments contain spaces that need to be passed on literally to the spawn call, they need to be quoted. As long as the subcomponents within the *path* arguments are quoted, using either of the following methods:

```
\'path with space\'
```

or

```
\"path with space\",
```

the following is allowed:

```
"\'path with space\' arg1 arg2 \"arg3 with space\"".
```

This would be parsed as

```
"path with space" -> path

arg1 = arg1

arg2 = arg2

arg3 = "arg3 with space".
```

The command line is in turn passed onto a *spawn* command by the HAM to create a new process that will execute the command.

The action is executed when the appropriate condition is triggered.

Note that this action also triggers a restart *condition* in the entity.

Currently the following flags are defined:

HREARMAFTERRESTART

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

HACTIONBREAKONFAIL

Indicates that if this action were to fail, and it is part of a list of actions, none of the actions following this one in the list of actions will be executed.

HACTIONKEEPONFAIL

Indicates that the action will be retained even if it fails. The default behavior is to remove failed actions. Nevertheless if the condition that this action is associated with is not retained, the action will get automatically removed.

Users can specify what will be done if an action fails. By adding action fail *actions* to a list associated with an action, users can determine what will be done in the case of an action failure. For every action that fails, the corresponding action fail list will be executed. Certain actions (e.g., *ham_action_log()* and *ham_action_heartbeat_healthy()*) never fail. For more details, refer to the appropriate section in the HAM API reference for the *ham_action_fail_*()* calls.

Returns:

A valid handle to an action in a condition, or NULL if an error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_restart()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*chdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_action_waitfor()

Add a waitfor action to a condition

Synopsis:

```
#include <ha/ham.h>

ham_action_t *ham_action_waitfor(
                  ham_condition_t *chdl,
                  const char *aname,
                  const char *path,
                  int delay,
                  unsigned flags);
```

Library:

`libham`

Description:

The *ham_action_waitfor()* function adds an action (*aname*) to a given condition. In this case, the action is one that lets you insert arbitrary delays into a sequence of actions.

The waitfor call fails if the specified path does not appear in the specified interval.

The handle (*chdl*) is obtained either:

- from one of the *ham_condition*()* functions to add conditions

  or:

- by calling any of the *ham_condition_handle()* functions to request a handle to a specific condition.

You use the *delay* parameter to specify the "waitfor" period in milliseconds.

You can also specify a *path* in order to control the delay. If *path* is specified, then the action waits for either the pathname to appear in the namespace or for the period specified in *delay*, whichever is *lower*.

Currently the following flag is defined:

HREARMAFTERRESTART

Indicates that the action is to be automatically rearmed after the entity that it belongs to is restarted. By default, this flag is *disabled* — actions automatically get pruned across restarts of the entity. Note that if the condition that this action belongs to is pruned after a restart, this action will also be removed, regardless of the value of this flag.

Returns:

A valid handle to an action to a condition, or NULL if an error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The name given in *aname* is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_action_waitfor()* aren't the same.

The delay specified is invalid. Only values greater than zero are permitted.

The condition into which the action is being added has the `HCONDNOWAIT` set.

`EEXIST`

The name provided for the action already exists.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity or condition specified by the given handle (*chdl*).

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

ham_attach(), ham_attach_node()

Attach an entity

Synopsis:

```
#include <ha/ham.h>

ham_entity_t *ham_attach( const char *ename,
                          int nd,
                          pid_t pid,
                          const char *line,
                          unsigned flags );

ham_entity_t *ham_attach_node( const char *ename,
                               const char *nodename,
                               pid_t pid,
                               const char *line,
                               unsigned flags );
```

Library:

```
libham
```

Description:

You use the *ham_attach()* function to attach an *entity* to the HAM. The *ham_attach_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*). An entity can be any process on the system. You can use this function to:

- attach a process that's already running

  or:

- tell the HAM to start a process and then add it as an entity to its context.

Once an entity has been attached, you can add *conditions* and *actions* to it. For arbitrary processes, the HAM can monitor either:

- processes that are in session 1 (e.g., by calling *procmgr_daemon()*)

  or:

- any process that dies due to the delivery of a core-dump signal, i.e., one of the set of signals that causes a core-dump. For more information on these signals, refer to the dumper utility in the *Utility Reference*.

Since the *ham_attach*()* functions open a connection to the HAM, for convenience they also perform the initial *ham_connect()* (p. 134) call. So any client that makes a *ham_attach()* call doesn't need to call *ham_connect()* (p. 134) or *ham_disconnect()* (p. 142) before and after the call to *ham_attach()*.

The arguments are as follows:

ename

The name of the entity; it must be unique across the whole context of the HAM.

nd

This *ham_attach()* parameter specifies the node identifier of the remote node on which the entity being targeted is running (or will be run). The *nd* is the node identifier of the remote node at the time the call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

pid

Process ID to attach to, if the process is already running. If *pid*   0, the HAM starts the process and begins monitoring it. In this case, *line* must also be specified with the FULL *path* (including all required arguments) to start the process.

nodename

This *ham_attach_node()* parameter is a fully qualified node name (FQNN).

line

This contains the FULL command line, including arguments, to start the process. This is used ONLY if *pid*   0 and is ignored otherwise. If either the pathname or the arguments contain spaces that need to be passed literally to the spawn call, they need to be quoted. As long as the subcomponents within the *path* arguments are quoted, using either of the following methods:

```
\'path with space\'
```

or:

```
\"path with space\",
```

the following is allowed:

```
"\'path with space\' arg1 arg2 \"arg3 with space\"".
```

This would be parsed as:

```
"path with space" -> path
arg1 = arg1
arg2 = arg2
arg3 = "arg3 with space".
```

flags

Currently, the following flag is defined:

`HENTITYKEEPONDEATH`

Indicates that the entity and all it conditions will be retained when the entity dies and is not restarted. The default is to remove all entities that aren't restarted.

Returns:

A valid handle to an entity on success; otherwise, NULL (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EEXIST`

The name provided for the entity already exists.

`EINVAL`

The name given in *ename* is invalid (e.g., it contains the “/” character) or is NULL.

The *pid* provided is 0, but no *line* was provided.

`ENAMETOOLONG`

The name given (in *ename*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing the request to add a new entity to its context.

Classification:

QNX Neutrino

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

ham_attach_self()

Attach an application as a self-attached entity

Synopsis:

```
#include <ha/ham.h>

ham_entity_t *ham_attach_self(
                  const char *ename,
                  _Uint64t hp,
                  int hpdl,
                  int hpdh,
                  unsigned flags );
```

Library:

`libham`

Description:

You use the *ham_attach_self()* call to attach an application as a *self-attached* entity to the HAM. A self-attached entity is a process that chooses to send heartbeats to the HAM, which will then monitor the process for failure. Self-attached entities can, on their own, decide at exactly what point in their lifespan they want to be monitored, what conditions they want acted upon, and when they want to stop the monitoring.

Note that self-attached entities can be any processes, not just those in session 1 (unlike the requirement for the *ham_attach()* (p. 110) call).

Once an entity has been attached, you can add *conditions* and *actions* to it.

Since the *ham_attach*()* functions open a connection to the HAM, for convenience they also perform the initial *ham_connect()* (p. 134) call. So any client that makes a *ham_attach_self()* call doesn't need to call *ham_connect()* (p. 134) or *ham_disconnect()* (p. 142) before and after the call to *ham_attach_self()*.

The arguments are as follows:

ename

The name of the entity. This must be unique across the whole context of the HAM. Self-attached entities can also specify an interval at which they'll send heartbeats to the HAM. The heartbeat can be used to detect unresponsive processes that aren't dead.

hp

The heartbeat interval in nanoseconds. The lowest permissible heartbeat interval is defined in the constant `HAMHBEATMIN` (see `<ha/ham.h>`). Use 0 if no heartbeat is desired.

Note that here you're specifying the heartbeat interval — the client must still call the *ham_heartbeat()* function to actually *transmit* the heartbeat.

hpdl

The number of permissible missed heartbeats before `CONDHBEATMISSEDLOW` is triggered. The value of *hpdl* must be *hpdh*.

hpdh

As for *hpdl*, but for `CONDHBEATMISSEDHIGH`. The value of *hpdh* must be *hpdl*.

flags

There are no flags defined at this time.

Returns:

A valid handle to an entity on success; otherwise, NULL (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EEXIST`

The name provided for the entity already exists.

`EINVAL`

The name given in *ename* is invalid (e.g., it contains the "/" character) or is NULL.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e. it exceeds _POSIX_PATH_MAX (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by _POSIX_PATH_MAX.

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing the request to add a new entity to its context.

Classification:

QNX Neutrino

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

ham_condition()

Set up a condition to be triggered when a certain event occurs

Synopsis:

```
#include <ha/ham.h>

ham_condition_t *ham_condition(ham_entity_t *ehdl,
                               int type,
                               const char *cname,
                               unsigned flags);
```

Library:

`libham`

Description:

You call this function to set up a condition that will be triggered when a significant event occurs.

Each entity can be associated with several different conditions, and associated with each of these conditions is a set of actions that will be performed in FIFO sequence when the condition is true. If an entity has multiple conditions that are true simultaneously, with different sets of actions associated with each condition, then all the actions associated with each condition are performed, in sequence.

This mechanism lets you combine actions together into sets and choose to remove/control them as a single “group” instead of as individual items.

The *ehdl* argument is an entity handle obtained either:

- from a *ham_attach*()* call or a *ham_entity*()* call

  or:

- by calling *ham_entity_handle*()* with an existing entity name. Since conditions are associated with entities, the entity handle (*ehdl*) must be available before you can add conditions.

Since conditions are associated with entities, the entity handle (*ehdl*) must be available before you can add conditions.

You can specify any of the following for *type*:

CONDDEATH

The entity died.

An entity's death is detected for all processes in session 1, for processes that terminate abnormally (typically due to the delivery of a signal), and for processes that are attached as self-attached entities.

CONDDETACH

The entity detached from the HAM.

CONDHBEATMISSEDLOW

The entity missed a heartbeat specified as “low” severity.

CONDHBEATMISSEDHIGH

The entity missed a heartbeat specified as “high” severity.

CONDRESTART

The entity was restarted.

The *cname* argument is the condition name. It must be unique across all conditions in the given entity.

When a condition is triggered, all actions defined in all conditions of the given type are executed. When an entity dies, a condition of type `HCONDDEATH` is triggered, and all actions in all conditions that match this type are executed.

Currently the following flags are defined:

HCONDINDEPENDENT

Indicates that the actions associated with this condition are to be performed in a separate thread. When a condition is triggered, actions within it are performed in FIFO order. For multiple conditions that are simultaneously triggered, the conditions are serviced in an arbitrary order. By setting this flag, you're marking the condition as *independent* — all actions associated with it are executed in a separate thread, independent of actions in other conditions.

HCONDNOWAIT

Indicates that the condition can't contain any “waitfor” actions. Waitfor actions are normally slow and may contain significant delays. This will delay the execution of subsequent actions in the list. Specifying `HCONDNOWAIT` guarantees there will be no delays once the condition is triggered.

HREARMAFTERRESTART

Indicates that the condition is to be automatically re-armed after the entity that it belongs to is restarted. Be default, this flag is disabled — conditions automatically get removed across restarts of the entity. Note that if the entity that the condition belongs to gets removed, this condition will also be removed, regardless of the state of this flag.

You must call the *ham_connect()* function *before* the first call to *ham_condition()* (p. 117) in a process. If a process calls *ham_connect()* (p. 134) and then calls *fork()*, the child process must call *ham_connect()* (p. 134) again before it can successfully call *ham_condition()* in order to add conditions.

Returns:

A valid handle to a condition in the given entity; otherwise, `NULL` (and *errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EEXIST`

The name provided for the condition already exists.

`EINVAL`

The handle, type, or name given is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_condition()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_entity_control()

Perform control operations on an entity object in a HAM

Synopsis:

```
#include <ha/ham.h>

int ham_entity_control( ham_entity_t *ehdl,
                        int command,
                        unsigned flags);
```

Library:

```
libham
```

Description:

The *ham_entity_control()* function can be used to control the state of an entity object in a HAM. This function is designed to be extensible with additional commands. Currently, the following commands are defined:

```
HENABLE,                 /* enable item             */
HDISABLE,                /* disable item            */
HADDFLAGS,               /* add flag                */
HREMOVEFLAG,             /* remove flag             */
HSETFLAGS,               /* set flag to specific    */
HGETFLAGS,               /* get flag                */
```

When an entity item is enabled (the default), any event that occurs in relation to this event will trigger appropriate conditions and actions related to the entity. If an entity item is disabled, no events relating to that entity will be *reacted to*. If an entity is disabled, all conditions and actions under it are also disabled. Individual conditions and actions can be enabled and disabled using the corresponding control functions for conditions and actions, respectively.

The HADDFLAGS, HREMOVEFLAGS, and HSETFLAGS commands can be used to modify the set of flags associated with the entity being controlled. Add flags and remove flags are used to either add to or remove from the current set of flags, the specified set of flags (as given in *flags*). The set flags function is called when the current set of flags is to be replaced by *flags*.

Flags

Any flag that is valid for the corresponding condition can be used when *ham_condition_control()* is being used to set flags, with the exception of HCONDNOWAIT if the existing condition already has some waitfor actions associated with it.

For the HENABLE and HDISABLE commands:

HRECURSE

Applies the command recursively.

Returns:

For enable, disable, add flags, remove flags, and set flags functions:

0

Success.

-1

An error occurred (*errno* is set).

For get flags function

flags

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The *command* or *flags* variable is invalid.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_condition_handle(), ham_condition_handle_node()

Get a handle to a condition in an entity

Synopsis:

```
#include <ha/ham.h>

ham_condition_t *ham_condition_handle(
                    int nd,
                    const char *ename,
                    const char *cname,
                    unsigned flags );

ham_condition_t *ham_condition_handle_node(
                    const char *nodename,
                    const char *ename,
                    const char *cname,
                    unsigned flags );
```

Library:

`libham`

Description:

The *ham_condition_handle()* function returns a handle to a condition (*cname*) in an entity (*ename*).

The handle obtained from this function can be passed to other functions, such as *ham_action_restart()* (p. 104) or *ham_condition_handle_free()* (p. 125).

To get a handle for a global entity, pass `NULL` for *ename*.

The handle returned is opaque; its contents are internal to the library.

If a node (*nd*) is specified, the handle is to an entity/condition/action combination that refers to a process on that remote node. The *ham_condition_handle_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*).

There are no flags defined at this time.

Returns:

A valid `ham_condition_t`, or `NULL` if an error occurred (*errno* is set).

Errors:

EINVAL

The name given in *cname* or *ename* is invalid (e.g., it contains the "`/`" character), or *cname* is `NULL`.

ENAMETOOLONG

The name given (in *aname*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

ENOENT

There's no condition by the name *cname* defined in an entity by name *ename* in the current context of the HAM.

ENOMEM

Not enough memory to create a new handle.

Classification:

QNX Neutrino

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

Caveats:

A call to *ham_condition_handle()* and a subsequent use of the handle returned in a call such as *ham_action_restart()* are completely *asynchronous*. Therefore, a valid action/condition/entity may no longer exist when the handle is used to attach actions at a later time.

In such an event, the *ham_action*()* functions will return an error (`ENOENT`) that the condition doesn't exist in the given entity.

ham_condition_handle_free()

Free a previously obtained handle to a condition in an entity

Synopsis:

```
#include <ha/ham.h>

int ham_condition_handle_free( ham_condition_t *chdl );
```

Library:

`libham`

Description:

The *ham_condition_handle_free()* function frees a handle associated with a condition (*chdl*) and reclaims all storage associated with the given handle.

The handle *chdl* must be obtained from *ham_condition_handle()* (p. 123) or *ham_condition()* (p. 117). Once a handle is freed, it is no longer available to refer to any condition. The *ham_condition_handle_free()* call frees storage allocated for the handle itself but does not remove the condition itself, which is in the HAM.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EINVAL`

The handle passed to the function isn't valid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The *ham_condition_handle_free()* function frees storage related only to the handle in the client — it doesn't remove the corresponding entity.

ham_condition_raise()

Attach a condition associated with a condition raise condition

Synopsis:

```
#include <ha/ham.h>

ham_condition_t *ham_condition_raise(
                             ham_entity_t *ehdl,
                             const char *cname,
                             unsigned rtype,
                             unsigned rclass,
                             unsigned rseverity,
                             unsigned flags );
```

Library:

`libham`

Description:

This condition is triggered whenever an entity raises a condition, which matches the given *rtype*, *rclass*, and *rseverity*. An entity that raises a condition, does so with a given set of values for type, class, and severity. Subscribers to this event can specify the conditions they are interested in either explicitly or by using the following special wild cards for each of these values.

```
CONDRAISETYPEANY     /* ANY type     : raised condition */
CONDRAISECLASSANY    /* ANY class    : raised condition */
CONDRAISESEVERITYANY /* ANY severity : raised condition */
```

The *ehdl* argument is an entity handle obtained either:

- from a *ham_attach*()* call or a *ham_entity*()* call

  or:

- by calling *ham_entity_handle*()* with an existing entity name. Since conditions are associated with entities, the entity handle (*ehdl*) must be available before you can add conditions.

Returns:

A condition handle, or `NULL` if an error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EEXIST

The name provided for the condition already exists.

EINVAL

The handle, type, or name given is invalid (e.g., it contains the “/” character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_condition()* aren't the same.

ENAMETOOLONG

The name given (in *aname*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

ENOMEM

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_condition_remove()

Remove a condition from an entity

Synopsis:

```
#include <ha/ham.h>

int ham_condition_remove( ham_condition_t *chdl,
                          unsigned flags );
```

Library:

`libham`

Description:

The *ham_condition_remove()* function removes a condition from an entity.

The *chdl* argument is a handle to a condition that was previously obtained by a call to *ham_condition()* or to *ham_condition_handle()* (p. 123).

There are no flags defined at this time.

The *ham_connect()* (p. 134) function must be called before the first call to *ham_condition_remove()* in a process. If a process calls *ham_connect()* (p. 134) and then calls *fork()*, the child process needs to call *ham_connect()* (p. 134) again before it can successfully call *ham_condition_remove()* to remove conditions.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The handle passed as an argument is invalid.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_condition_remove()* aren't the same.

ENOENT

There's no condition corresponding to the handle supplied.

In addition to the above errors, the HAM returns any error it encounters while servicing the request.

Classification:

QNX Neutrino

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

ham_condition_state()

Attach a condition associated with a state transition

Synopsis:

```
#include <ha/ham.h>

ham_condition_t *ham_condition_state(
                         ham_entity_t *ehdl,
                         const char *cname,
                         unsigned fromstate,
                         unsigned tostate,
                         unsigned flags );
```

Library:

`libham`

Description:

This condition is triggered when the specified entity changes from a state matching *fromstate* to a state matching *tostate*. `CONDSTATEANY` can be used to specify a wild card signifying any STATE. The matching of states is based on either an explicit match or special conditions explained in the Flags section below.

The *ehdl* argument is an entity handle obtained either:

- from a *ham_attach*()* call or a *ham_entity*()* call

  or:

- by calling *ham_entity_handle*()* with an existing entity name. Since conditions are associated with entities, the entity handle (*ehdl*) must be available before you can add conditions.

Flags

Standard flags that are applicable for conditions are also valid here. (See *ham_condition()* (p. 117) for more details). In addition, the following flags are also defined:

```
/* special condition flags */

HCONDSTATEFROMSET /* from state is a set */
HCONDSTATETOSET /* to state is a set */
HCONDSTATEFROMINV /* invert from state set */
HCONDSTATETOINV /* invert to state set */
```

States can be given using:

- individual values

- the wild card `CONDSTATEANY`
- bitmaps that allow several states to be referred to collectively as sets.

If *fromstate* or *tostate* refers to a set, the corresponding flag `HCONDSTATEFROMSET` or `HCONDSTATETOSET` must be set.

If you want *fromstate* or *tostate* to refer to the logical negation of a set of states define `HCONDSTATEFROMINV` or `HCONDSTATETOINV` in flags. This would logically invert the set of states specified (equivalent to saying "any state other than this set").

Returns:

A condition handle, or `NULL` if an error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EEXIST`

The name provided for the condition already exists.

`EINVAL`

The handle, type, or name given is invalid (e.g., it contains the "`/`" character) or is NULL.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling *ham_condition()* aren't the same.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_connect(), ham_connect_nd(), ham_connect_node()

Connect to the HAM

Synopsis:

```
#include <ha/ham.h>

int ham_connect( unsigned flags);
int ham_connect_nd( int nd, unsigned flags);
int ham_connect_node( const char *nodename, unsigned flags);
```

Library:

`libham`

Description:

The *ham_connect()* function initializes a connection to a HAM. The *ham_connect_nd()* and *ham_connect_node()* functions initialize connections to remote HAMs.

A process may have only a single connection open to a HAM at any time.

The *nd* specified to `ham_connect_nd` is the node identifier of the remote node at the time the *ham_connect_nd()* call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

The *ham_connect_node()* function takes as a parameter a fully qualified node name (FQNN).

You can call these functions any number of times, but because the library maintains a reference count, you need to call *ham_disconnect()* (p. 142) the same number of times to release the connection.

Connections are associated with a specific process ID (*pid*). If a process performs *ham_connect()* and then calls *fork()*, the child process needs to reconnect to the HAM by calling *ham_connect()* again.

But if a process calls any of the following:

- *ham_attach_self()* (p. 114)

- *ham_attach()* (p. 110)
- *ham_attach_node()* (p. 110)
- *ham_detach()* (p. 136)
- *ham_detach_self()* (p. 140)
- *ham_detach_name()* (p. 138)
- *ham_detach_name_node()* (p. 138)
- *ham_stop()* (p. 159)
- *ham_stop_nd()* (p. 159)
- *ham_stop_node()* (p. 159)
- *ham_entity()* (p. 144)
- *ham_entity_node()* (p. 144)
- *ham_verbose()* (p. 161)

it doesn't need to call *ham_connect*()*, since those functions do so on their own.

For all other *ham*()* functions, clients must call *ham_connect()* first.

There are no flags defined at this time.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

Upon failure, the *ham_connect*()* functions return the error as set by the underlying *open()* library call that failed.

Classification:

QNX Neutrino

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

ham_detach()

Detach an entity from the HAM

Synopsis:

```
#include <ha/ham.h>

int ham_detach( ham_entity_t *ehdl,
                unsigned flags );
```

Library:

`libham`

Description:

This function detaches an attached entity from the HAM. The entity being detached must *not* be a self-attached entity.

The handle passed into this function (*ehdl*) is either returned by a previous *ham_attach()* (p. 110) call or obtained from *ham_entity_handle()* (p. 153).

There are no flags defined at this time.

This function automatically calls *ham_connect()* (p. 134) and *ham_disconnect()* (p. 142) for the client.

The *ham_detach()* call automatically raises a `HCONDDETACH` condition in the HAM for that entity.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The handle passed as an argument is invalid.

ENOENT

There's no entity corresponding to the handle supplied.

In addition to the above errors, the HAM returns any error it encounters while servicing the request to remove the entity from its context.

Classification:

QNX Neutrino

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

ham_detach_name(), ham_detach_name_node()

Detach an entity from the HAM, using an entity name

Synopsis:

```
#include <ha/ham.h>

int ham_detach_name( int nd,
                     const char *ename,
                     unsigned flags);

int ham_detach_name_node( const char *nodename,
                     const char *ename,
                     unsigned flags);
```

Library:

`libham`

Description:

These functions detach an attached entity (*ename*) from a HAM. The entity being detached must NOT be a self-attached entity.

The *nd* specifies the node identifier of the remote node on which the entity being targeted is running, at the time the call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

The *ham_detach_name_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*).

There are no flags defined at this time.

This function automatically calls *ham_connect()* (p. 134) and *ham_disconnect()* (p. 142) for the client.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Error connecting to the HAM.

`EINVAL`

The name passed as an argument is invalid.

`ENOENT`

There's no entity corresponding to the name supplied.

In addition to the above errors, the HAM returns any error it encounters while servicing the request to remove the entity from its context.

Classification:

QNX Neutrino

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

ham_detach_self()

Detach a self-attached entity from the HAM

Synopsis:

```
#include <ha/ham.h>

int ham_detach_self( ham_entity_t *ehdl
                     unsigned flags );
```

Library:

`libham`

Description:

This function detaches a self-attached entity from the HAM. The entity being detached *must* be a self-attached entity.

The handle passed into this function (*ehdl*) is either returned by a previous *ham_attach_self()* (p. 114) call or obtained from *ham_entity_handle()* (p. 153).

There are no flags defined at this time.

This function automatically calls *ham_connect()* (p. 134) and *ham_disconnect()* (p. 142) for the client.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Error connecting to the HAM.

EINVAL

The handle passed as an argument is invalid.

ENOENT

There is no entity corresponding to the handle supplied.

In addition to the above errors, the HAM returns any error it encounters while servicing the request to remove the entity from its context.

Classification:

QNX Neutrino

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

ham_disconnect(), ham_disconnect_nd(), ham_disconnect_node()

Disconnect from the HAM

Synopsis:

```
#include <ha/ham.h>

int ham_disconnect( unsigned flags);
int ham_disconnect_nd( int nd, unsigned flags);
int ham_disconnect_node( const char *nodename, unsigned flags);
```

Library:

`libham`

Description:

The *ham_disconnect()* function closes a previously open connection to a HAM. The *ham_disconnect_nd()* and *ham_disconnect_node()* functions close previously opened connections to remote HAMs.

The *nd* specified to `ham_disconnect_nd` is the node identifier of the remote node at the time the *ham_disconnect_nd()* call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

The *ham_disconnect_node()* function takes as a parameter a fully qualified node name (FQNN).

Because the library maintains a reference count, the actual connection to the HAM is released only when the number of calls made to *ham_disconnect()* matches the number of calls previously made to *ham_connect()* (p. 134).

When a process calls *ham_connect()* (p. 134) and then calls *fork()*, the connection is no longer valid in the child process. To reconnect to the HAM, the child process must call either:

- *ham_connect()* (p. 134) directly, which will automatically close the existing connection and reopen a new one

  or:

- *ham_disconnect()*, followed by *ham_connect()* (p. 134).

There are no flags defined at this time.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EINVAL`

There's no valid connection to the HAM to disconnect.

Classification:

QNX Neutrino

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

ham_entity(), ham_entity_node()

Create entity placeholder objects in a HAM

Synopsis:

```
#include <ha/ham.h>

ham_entity_t *ham_entity( const char *ename,
                          int nd,
                          unsigned flags);

ham_entity_t *ham_entity_node( const char *ename,
                          const char *nodename,
                          unsigned flags);
```

Library:

`libham`

Description:

These functions are used to create placeholders for entity objects in a HAM. The *ham_entity_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*).

You can use these functions to create entities, and associate conditions and actions with them, before the process associated with an entity is started (or attached). Subsequent *ham_attach*()* calls by entities can attach to these placeholder and thereby enable conditions and actions when they occur.

The *nd* variable specifies the node identifier of the remote node at the time the call is made.

Since node identifiers are transient objects, we recommend that the value for *nd* is obtained at the time of the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

The *ham_entity_node()* function takes as a parameter a fully qualified node name (FQNN).

Returns:

An entity handle, or `NULL` if an error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

`EINVAL`

The name given in *ename* is invalid (e.g., it contains the "`/`" character) or is NULL.

`ENAMETOOLONG`

The name given (in *aname*) is too long, i.e., it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOMEM`

Not enough memory to create a new handle.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_entity_condition_raise()

Used by an entity to raise a condition

Synopsis:

```
#include <ha/ham.h>

int ham_entity_condition_raise(
                         ham_entity_t *ehdl,
                         unsigned rtype,
                         unsigned rclass,
                         unsigned rseverity,
                         unsigned flags );
```

Library:

`libham`

Description:

This function is used by an entity to notify a HAM of an interesting event of its choice. This in turn triggers a `CONDITION_RAISE` in the HAM, which will search for matching subscribers for this event and execute all associated actions.

The values of *rtype*, *rclass*, and *rseverity* can be used to permit finer grain matching and to gather additional information relating to the condition.

To learn more about the matching mechanism, refer to the API documentation for *ham_condition_raise()* (p. 127).

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The handle specified in *ehdl* is invalid.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling this function are not the same.

ENOENT

There's no entity by the given handle (*ehdl*).

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_entity_condition_state()

Used by an entity to notify the HAM of a state transition

Synopsis:

```
#include <ha/ham.h>

int ham_entity_condition_state(
                         ham_entity_t *ehdl,
                         unsigned tostate,
                         unsigned flags );
```

Library:

`libham`

Description:

This function enables an entity to report a transition to the HAM; the value *tostate* indicates the transitional state. The HAM in turn triggers a condition state event for this entity, and will search for matching subscribers for this event and execute all associated actions. For more details of the matching mechanisms refer to the API documentation for *ham_condition_state()* (p. 131).

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

EINVAL

The handle specified in *ehdl* is invalid.

The connection to the HAM is invalid. This happens when the process that opened the connection (using *ham_connect()* (p. 134)) and the process that's calling this function are not the same.

ENOENT

There's no entity by the given handle (*ehdl*).

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_entity_control()

Perform control operations on an entity object in a HAM

Synopsis:

```
#include <ha/ham.h>

int ham_entity_control( ham_entity_t *ehdl,
                        int command,
                        unsigned flags);
```

Library:

`libham`

Description:

The *ham_entity_control()* function can be used to control the state of an entity object in a HAM. This function is designed to be extensible with additional commands. Currently, the following commands are defined:

`HENABLE`

Enable the item.

`HDISABLE`

Disable the item.

`HADDFLAGS`

Add the flags.

`HREMOVEFLAGS`

Remove the flags.

`HSETFLAGS`

Set the flags to the given value.

`HGETFLAGS`

Get the current flags.

When an entity item is enabled (the default), any event that occurs in relation to this event will trigger appropriate conditions and actions related to the entity. If an entity item is disabled, no events relating to that entity will be *reacted to*. If an entity is disabled, all conditions and actions under it are also disabled. Individual conditions and actions can be enabled and disabled using the corresponding control functions for conditions and actions, respectively.

The `HADDFLAGS`, `HREMOVEFLAGS`, and `HSETFLAGS` commands can be used to modify the set of flags associated with the entity being controlled. Add flags and remove flags are used to either add to or remove from the current set of flags, the specified set of flags (as given in *flags*). The set flags function is called when the current set of flags is to be replaced by *flags*.

Flags

Any flag that is valid for the corresponding entity can be used when *ham_entity_control()* is being used to set flags.

For the `HENABLE` and `HDISABLE` commands:

`HRECURSE`

Applies the command recursively.

Returns:

For enable, disable, add flags, remove flags, and set flags functions:

0

Success.

-1

An error occurred (*errno* is set).

For get flags function

flags

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The *command* or *flags* variable is invalid.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

ham_entity_handle(), ham_entity_handle_node()

Get a handle to an entity

Synopsis:

```
#include <ha/ham.h>

ham_entity_t *ham_entity_handle( int nd,
                                 const char *ename,
                                 unsigned flags );

ham_entity_t *ham_entity_handle_node( const char *nodename,
                                      const char *ename,
                                      unsigned flags );
```

Library:

`libham`

Description:

The *ham_entity_handle()* function returns a handle to an entity (*ename*). The handle can then be passed to other functions that expect a handle to an entity (such as *ham_condition()* (p. 117) or *ham_entity_handle_free()* (p. 155)).

To get a handle for a global entity, pass `NULL` for *ename*.

The handle returned is opaque; its contents are internal to the library.

If a node (*nd*) is specified, the handle is to an entity/condition/action combination that refers to a process on that remote node. The *ham_entity_handle_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*).

There are no flags defined at this time.

Returns:

A valid `ham_entity_t`, or `NULL` if an error occurred (*errno* is set).

Errors:

`EINVAL`

The name given in *ename* is invalid (e.g., it contains the "`/`" character).

`ENAMETOOLONG`

The name given (in *ename*) is too long, i.e. it exceeds `_POSIX_PATH_MAX` (defined in `<limits.h>`). Note that the *combined length* of an entity/condition/action name is also limited by `_POSIX_PATH_MAX`.

`ENOENT`

There's no entity by this name defined in the current context of the HAM.

`ENOMEM`

Not enough memory to create a new handle.

Classification:

QNX Neutrino

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

Caveats:

A call to *ham_entity_handle()* and a subsequent use of the handle returned in a call such as *ham_condition()* are completely *asynchronous*. Therefore, a valid action/condition/entity may no longer exist when the handle is used at a later time.

In such an event, the *ham_condition*()* functions will return an error (`ENOENT`) that the action in the condition doesn't exist in the given entity.

ham_entity_handle_free()

Free a previously obtained handle to an entity

Synopsis:

```
#include <ha/ham.h>

int ham_entity_handle_free( ham_entity_t *ehdl );
```

Library:

`libham`

Description:

The *ham_entity_handle_free()* function frees a handle associated with an entity (*ehdl*) and reclaims all storage associated with the given handle.

The handle (*ehdl*) must be obtained from *ham_entity_handle()* (p. 153), *ham_attach()* (p. 110), or *ham_attach_self()* (p. 114). Once a handle is freed, it is no longer available to refer to any condition. The *ham_entity_handle_free()* call frees storage allocated for the handle but does not remove the condition itself, which is in the HAM.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EINVAL`

The name given in *ename* isn't valid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

The *ham_entity_handle_free()* function frees storage related only to the handle in the client — it doesn't remove the corresponding entity.

ham_heartbeat()

Send a heartbeat to the HAM

Synopsis:

```
#include <ha/ham.h>

int ham_heartbeat( void );
```

Library:

`libham`

Description:

Self-attached entities that have committed to sending heartbeats at prescribed intervals need to call *ham_heartbeat()* when they want to transmit a heartbeat.

The *ham_heartbeat()* function does nothing if the client isn't a self-attached entity or hasn't committed to sending heartbeats.

Returns:

This function always succeeds.

Classification:

QNX Neutrino

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

Caveats:

Although this function always succeeds, the HAM doesn't always receive the heartbeat right away.

For example, if a client commits to sending a heartbeat every 5 seconds (at 5-, 10-, 15-second intervals, and so on), but instead transmits at the 2-second mark, then the HAM won't receive a heartbeat until the 5-second mark.

Or if the client sends a heartbeat at the 7-second mark and another at the 8-second mark, then the HAM will receive only *one heartbeat* at the 10-second mark.

ham_stop(), ham_stop_nd(), ham_stop_node()

Stop the HAM

Synopsis:

```
#include <ha/ham.h>

int ham_stop( void );
int ham_stop_nd( int nd);
int ham_stop_node( const char *nodename);
```

Library:

`libham`

Description:

The *ham_stop()* function instructs the HAM to terminate. The *ham_stop_nd()*, and *ham_stop_node()* functions are used to terminate remote HAMs. These are the only proper ways to stop the HAM.

The *nd* specified to *ham_stop_nd()* is the node identifier of the remote node at the time the *ham_stop_nd()* call is made.

Since node identifiers are transient objects, you should obtain the value for *nd* immediately before the call, using *netmgr_strtond()* or another function that converts nodenames into node identifiers.

The *ham_stop_node()* function takes as a parameter a fully qualified node name (FQNN). The *ham_stop_node()* function is used when a nodename is used to specify a remote HAM instead of a node identifier (*nd*).

Since the HAM and its “clone” the Guardian monitor each other, and re-spawn should the other fail, the HAM must first terminate the Guardian before it terminates itself.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

EBADF

Couldn't connect to the HAM.

In addition to the above, the HAM returns any error it encounters while servicing the request to terminate.

Classification:

QNX Neutrino

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

ham_verbose()

Modify the verbosity of a HAM

Synopsis:

```
#include <ha/ham.h>

int ham_verbose( const char *nodename,
                 int op,
                 int value);
```

Arguments:

nodename

The name of the node on which to make the change, or `NULL` if you want to change the verbosity on the current node.

op

The operation to perform on the verbosity; one of the following:

- `VERBOSE_SET_INCR` — increment the verbosity.
- `VERBOSE_SET_DECR` — decrement the verbosity.
- `VERBOSE_SET` — set the verbosity to a specific value.
- `VERBOSE_GET` — get the verbosity.

value

The increment, decrement, or specific value for the verbosity, depending on the value of *op*. The *value* must be a non-negative integer. If *value* is zero, the function uses 1 for the increment or decrement.

Library:

`libham`

Description:

The *ham_verbose* function can be used to get or modify the verbosity of a HAM. You can also use the `hamctrl` utility to do this.

Returns:

When setting the verbosity:

0

Success.

-1

An error occurred (*errno* is set).

If *op* is `VERBOSE_GET`, the function returns the current verbosity, or -1 if an error occurred (*errno* is set).

Errors:

`EBADF`

Couldn't connect to the HAM.

`EINVAL`

The *value* or *op* variable is invalid.

In addition to the above errors, the HAM returns any error it encounters while servicing this request.

Classification:

QNX Neutrino

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

Chapter 7
Client Recovery Library Reference

The Client Recovery Library includes the following convenience functions you can use in your applications to transparently restore client/server connections in the event of failures:

Function	Description
ha_attach() (p. 164)	Attach a recovery function to a connection to make it HA-aware.
ha_close() (p. 167)	Detach an attached HA *fd*, then close it.
ha_connection_ctrl() (p. 168)	Control the operation of a HA-aware connection.
ha_ConnectAttach(), ha_ConnectAttach_r() (p. 170)	Create a connection using *ConnectAttach()* and attach it to the HA lib.
ha_ConnectDetach(), ha_ConnectDetach_r() (p. 172)	Detach an attached *fd*, then close the connection using *ConnectDetach()*.
ha_creat(), ha_creat64() (p. 174)	Create a connection and attach it to the HA lib.
ha_detach() (p. 176)	Detach a connection previously attached via *ha_attach()*.
ha_dup() (p. 178)	Duplicate an HA connection.
ha_fclose() (p. 180)	Detach an attached HA *fd* for a file stream, then close it.
ha_fopen() (p. 181)	Open a file stream and attach it to the HA lib.
ha_open(), ha_open64() (p. 183)	Open a connection and attach it to the HA lib.
ha_ReConnectAttach() (p. 185)	Reopen a connection while performing recovery.
ha_reopen() (p. 187)	Reopen a connection while performing recovery.

For information on using these *ha_*()* functions, see the chapter *Using the Client Recovery Library* (p. 55) in this guide.

ha_attach()

Attach a recovery function to a connection to make it HA-aware

Synopsis:

```
#include <ha/cover.h>

int ha_attach(int coid,
              RFp rfn,
              void *rhdl,
              unsigned flags);
```

Library:

```
libha
```

Description:

The *ha_attach()* function attaches a recovery function to a connection in order to make the connection identified by *coid* HA-aware. If any operation on the connection *coid* returns an EBADF error, the recovery function pointed to by *rfn* will be called. The recovery function is defined by the following type declaration in `<ha/types.h>`:

```
typedef int (*RFp)(int coid, void *rhdl);
```

The recovery function identified by *rfn* will be called with *rhdl* as a parameter. The *rhdl* parameter is an opaque handle to data that will be interpreted and used by the recovery function itself. The recovery function is expected to perform client-specific recovery on the existing connection.

The recovery function returns a connection ID associated with the recovered connection. This connection ID must be the same as the one that had failed. The client can choose to recover in any way it thinks appropriate. It could reconnect to the same server (if the service is available), and then reconstruct its state with respect to the connection as appropriate from the client's perspective.

The client could also reconnect to a new server. The client recovery function must return the same connection ID in order to indicate successful recovery to the HA library so that the library can re-initiate the previous failed operation on the connection.

If the client doesn't want to — or can't — recover, it can return -1 to the library. The library will then immediately propagate the error relating to the failed operation on the connection back to the caller. For convenience, the HA library provides *ha_reopen()* and *ha_ReConnectAttach()* calls that close the old connection and obtain the new connection appropriately.

You normally call *ha_attach()* after a connection is established and a valid *coid* is available.

The other method to make a connection HA-aware is to call the convenience functions *ha_open()*, *ha_creatl64()*, *ha_ConnectAttachl_r()*, or *ha_fopen()*.

Currently the following flag is defined:

HAREPLACERECOVERYFN

Indicates that the call to *ha_attach()* is replacing the recovery function with a new one. You can replace recovery functions only if the connection already has a recovery function associated with it.

Returns:

0

Success

-1

An error occurred (*errno* is set).

Errors:

EBADF

There's no connection identified by *coid*.

Or, HAREPLACERECOVERYFN is set, but there's no HA-aware connection identified by *coid*.

EEXIST

There's already an HA-aware connection identified by *coid*.

The flag HAREPLACERECOVERYFN isn't set.

ENOMEM

Memory couldn't be allocated while creating the structures in the library.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

ha_close()

Detach an attached HA file descriptor, then close it

Synopsis:

```
#include <ha/unistd.h>

int ha_close(int fd);
```

Library:

`libha`

Description:

The *ha_close()* convenience function detaches a connection that was previously attached using *ha_attach()*, and then closes the connection.

The *fd* is the file descriptor originally obtained from *ha_open()*.

Returns:

0

Success

-1

An error occurred (*errno* is set).

Errors:

The *ha_close()* function returns errors as returned by either the underlying *close()* call or the *ha_detach()* call.

Classification:

QNX Neutrino

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

ha_connection_ctrl()

Control the operation of a HA-aware connection

Synopsis:

```
#include <ha/cover.h>
int ha_connection_ctrl( int coid,
                        int command,
                        void *args );
```

Library:

`libha`

Description:

The *ha_connection_ctrl()* function can be used to control the operation of the HA-aware conenction. Specifically it can be used to control the recovery method, temporarily suspend recovery and also control the number of times that will be attempted both consecutively for a single failure, or across all failures. Currently the following commands are defined:

HA_RECOVERY_ACOUNT

This sets the maximum number of times recovery is performed for this connection. The value is specified by passing it via “args” as an integer. A negative value implies that there is no limit on the number of times recovery will be performed, and this is the default state of the connection when it is made HA aware.

HA_RECOVERY_ICOUNT

This sets the maximum number of iterations, recovery is performed for this connection, each time a connection is found to have failed. This count is reset each time the connection is successfully recovered. The value is specified by passing it via “args” as an integer. A negative value implies that there is no limit on the number of times recovery will be performed, and this is the default state of the connection when it is made HA aware.

HA_RECOVERY_SUSPEND

Temporarily suspends any recovery on this connection. It will behave like a normal connection.

HA_RECOVERY_ENABLE

Re-enables any recovery on this connection. All other recovery options are unaffected. This just toggles the “recovery enabled/disabled” bit.

`HA_RECOVERY_RESET_ICOUNT`

Resets the iteration count used by `HA_RECOVERY_ICOUNT` above. This sets the internal count per iteration to zero.

`HA_RECOVERY_RESET_ACOUNT`

Resets the total count used by `HA_RECOVERY_ACOUNT` above. This sets the internal count of recoveries to zero.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Errors:

`EBADF`

No connection is specified by this *coid*, or connection is not HA-aware.

`EINVAL`

Invalid command.

Classification:

QNX Neutrino

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

ha_ConnectAttach(), ha_ConnectAttach_r()

Create a connection and attach it to the HA lib

Synopsis:

```
#include <ha/neutrino.h>

int ha_ConnectAttach(_Uint32t *nd,
                     pid_t pid,
                     int chid,
                     unsigned index,
                     unsigned flags,
                     RFp rfn,
                     void *rhdl,
                     unsigned haflags);

int ha_ConnectAttach_r(_Uint32t *nd,
                       pid_t pid,
                       int chid,
                       unsigned index,
                       unsigned flags,
                       RFp rfn,
                       void *rhdl,
                       unsigned haflags);
```

Library:

```
libha
```

Description:

The *ha_ConnectAttach()* and *ha_ConnectDetach_r()* functions are identical except in the way they return errors. (For details, see the "Returns" section.)

In addition to creating the connection using the standard *ConnectAttach|_r()* call, these convenience functions also call *ha_attach()* with the connection returned by the *ConnectAttach()* call.

The parameters *rfn()*, and *rhdl()*, and *haflags()* are passed to the *ha_attach()* call along with the connection ID returned by the *ConnectAttach()* call.

The remaining parameters are passed to the corresponding parameters in the *ConnectAttach()* call in their appropriate positions.

Returns:

The only difference between these functions is the way they indicate errors:

ha_ConnectAttach()

A connection ID that's used by the message primitives. If an error occurs, -1 is returned and *errno* is set.

ha_ConnectAttach_r()

A connection ID that's used by the message primitives. This function does NOT set *errno*. If an error occurs, the negative of a value from the errors returned by either the underlying *ConnectAttach()* call or the *ha_attach()* call.

Errors:

The *ha_ConnectAttachl_r()* call returns errors as returned by either the underlying *ConnectAttach()* call or the *ha_attach()* call.

Classification:

QNX Neutrino

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

ha_ConnectDetach(), *ha_ConnectDetach_r()*

Detach an attached file descriptor, then close the connection

Synopsis:

```
#include <ha/neutrino.h>

int ha_ConnectDetach(int coid);

int ha_ConnectDetach_r(int coid);
```

Library:

`libha`

Description:

The *ha_ConnectDetach()* and *ha_ConnectDetach_r()* functions are identical except in the way they return errors. (For details, see the "Returns" section.)

The *ha_ConnectDetach|_r()* call detaches a connection (*coid*) that was previously attached using *ha_attach()*, and then closes the connection by calling the appropriate *ConnectDetach|_r()* call.

Returns:

The only difference between these functions is the way they indicate errors:

ha_ConnectDetach()

If an error occurs, -1 is returned and *errno* is set. Any other value returned indicates success.

ha_ConnectDetach_r()

EOK is returned on success. This function does NOT set *errno*. If an error occurs, any value from the errors returned by either the underlying *ConnectDetach()* call or the *ha_detach()* call may be returned.

Errors:

The *ha_ConnectDetach|_r()* call returns errors as returned by either the underlying *ConnectDetach()* call or the *ha_detach()* call.

Classification:

QNX Neutrino

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

ha_creat(), ha_creat64()

Create a connection and attach it to the HA lib

Synopsis:

```
#include <ha/fcntl.h>

int ha_creat(const char *path,
             mode_t mode,
             RFp rfn,
             void *rhdl,
             unsigned haflags);

int ha_creat64(const char *path,
               mode_t mode,
               RFp rfn,
               void *rhdl,
               unsigned haflags);
```

Library:

`libha`

Description:

In addition to opening the connection using the standard *createl64()* call, these convenience functions also call *ha_attach()* with the connection returned by the *creat()* call.

The parameters *rfn()*, and *rhdl()*, and *haflags()* are passed to the *ha_attach()* call along with the connection ID returned by the *creat()* call.

The remaining parameters are passed to the corresponding parameters in the *creat()* call in their appropriate positions.

Returns:

A new connection ID or -1 if an error occurred (*errno* is set).

Errors:

The *ha_creat()* call returns errors as returned by either the underlying *creat()* call or the *ha_attach()* call.

Classification:

QNX Neutrino

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

ha_detach()

Detach a previously attached connection

Synopsis:

```
#include <ha/cover.h>

int ha_attach(int coid)
```

Library:

`libha`

Description:

The *ha_detach()* function detaches a connection from the HA library. This effectively makes the connection non HA-aware. After the detach operation is complete, no more recovery will be performed for any message operations on this connection.

The connection referred to by *coid* must be a connection previously attached to using *ha_attach()*. Normally, you detach connections just prior to closing them. The functions *ha_close()*, *ha_ConnectDetachl_r()*, and *ha_fclose()* perform the required *ha_detach()* operation before closing the connection.

Returns:

0

Success

-1

An error occurred (*errno* is set).

Errors:

EBADF

There's no connection *coid* that's currently attached.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ha_dup()

Duplicate an HA connection

Synopsis:

```
#include <ha/unistd.h>

int ha_dup(int oldfd);
```

Library:

`libha`

Description:

The *ha_dup()* function duplicates the HA-aware file descriptor specified by *oldfd*. The functionality of *ha_dup()* is similar to that of the standard *dup()* call, with the addition that the new file descriptor also shares the recovery mechanisms associated with *oldfd*.

Changing the recovery function for one file descriptor will automatically change the recovery function for the other as well.

Note that HA connections are also reference-counted across *ha_dup()* calls. This implies that when HA connections that have been *dup()*'d are closed, the recovery functions will continue to exist until the last reference to them has been closed.

Returns:

The new file descriptor or -1 if an error occurred (*errno* is set).

Errors:

`EBADF`

The connection identified by *oldfd* isn't an HA-aware connection.

`ENOMEM`

Couldn't allocate memory for structures in the library to successfully duplicate the connection.

In addition, the *ha_dup()* call returns any errors returned by the underlying *dup()* call.

Classification:

QNX Neutrino

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

ha_fclose()

Detach an attached HA file descriptor for a file stream, then close it

Synopsis:

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

int ha_fclose(FILE *stp);
```

Library:

`libha`

Description:

The *ha_fclose()* convenience function detaches a connection associated with the file stream *stp* that was previously attached using *ha_attach()*, and then closes the connection.

Returns:

0

Success

-1

An error occurred (*errno* is set).

Errors:

The *ha_fclose()* function returns errors as returned by either the underlying *fclose()* call or the *ha_detach()* call.

Classification:

QNX Neutrino

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

ha_fopen()

Open a file stream and attach it to the HA lib

Synopsis:

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

FILE *ha_fopen( const char *path,
                const char *mode,
                RFp rfn,
                void *rhdl,
                unsigned haflags);
```

Library:

`libha`

Description:

In addition to opening the connection using the standard *fopen()* call, this convenience function also calls *ha_attach()* with the connection returned by the *fopen()* call.

The parameters *rfn()*, and *rhdl()*, and *haflags()* are passed to the *ha_attach()* call along with the connection ID returned by the underlying *fopen()* call.

The remaining parameters are passed to the corresponding parameters in the *fopen()* call in their appropriate positions.

Returns:

A pointer to a file stream or `NULL` if an error occurs (*errno* is set).

Errors:

The *ha_fopen()* call returns errors as returned by either the underlying *fopen()* call or the *ha_attach()* call.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

ha_open(), ha_open64()

Open a connection and attach it to the HA lib

Synopsis:

```
#include <ha/fcntl.h>

int ha_open(const char *path,
            int oflag,
            RFp rfn,
            void *rhdl,
            unsigned haflags, ...);

int ha_open64(const char *path,
            int oflag,
            RFp rfn,
            void *rhdl,
            unsigned haflags, ...);
```

Library:

`libha`

Description:

In addition to opening the connection using the standard *openl64()* call, these convenience functions also call *ha_attach()* with the connection returned by the *open()* call.

The parameters *rfn()*, and *rhdl()*, and *haflags()* are passed to the *ha_attach()* call along with the connection ID returned by the *open()* call.

The remaining parameters are passed to the corresponding parameters in the *open()* call in their appropriate positions.

Returns:

A new connection ID or -1 if an error occurred (*errno* is set).

Errors:

The *ha_open()* call returns errors as returned by either the underlying *open()* call or the *ha_attach()* call.

Classification:

QNX Neutrino

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

ha_ReConnectAttach()

Reopen a connection while performing recovery

Synopsis:

```
#include <ha/neutrino.h>

int ha_ReConnectAttach(int oldcoid,
                       _Uint32t nd,
                       pid_t pid,
                       int chid,
                       unsigned index,
                       unsigned flags);
```

Library:

`libha`

Description:

You can use the *ha_ReConnectAttach()* convenience function to reopen a connection while in the recovery phase. The *oldcoid* argument refers to the connection that has failed. The *ha_ReConnectAttach()* function closes the previous connection and opens a new connection using the parameters specified by calling *ConnectAttach()*.

The *ha_ReConnectAttach()* function also verifies that the new connection ID returned is the same as the *oldcoid* (as required by the HA library mechanism).

Returns:

A new connection ID or -1 if an error occurred (*errno* is set).

Errors:

The *ha_ReConnectAttach()* call returns errors as returned by the underlying *ConnectAttach()* call.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

ha_reopen()

Reopen a connection while performing recovery

Synopsis:

```
#include <ha/fcntl.h>

int ha_reopen(int oldfd,
              const char *path,
              int oflag, ...);
```

Library:

`libha`

Description:

You can use the *ha_reopen()* convenience function to reopen a connection while in the recovery phase. The *oldfd* argument refers to the connection that has failed. The *ha_reopen()* function closes the previous connection and opens a new connection using the parameters specified by calling *open()*.

The *ha_reopen()* function also verifies that the new connection ID returned is the same as the *oldfd* (as required by the HA library mechanism). If the new connection ID obtained is not the same as *oldfd*, it will attempt to obtain the same *fd*, by calling the *dup2()* function.

Returns:

A new connection ID or -1 if an error occurred (*errno* is set).

Errors:

The *ha_reopen()* call returns errors as returned by the underlying *open()* call.

Classification:

QNX Neutrino

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

Chapter 8
Examples

This appendix includes sample code that handles various HA scenarios.

Simple restart

The most basic form of recovery is the simple death-restart mechanism. Since the QNX Neutrino RTOS provides virtually all non-kernel functionality via user-installable programs, and since it offers complete memory protection, not only for user applications, but also for OS components (device drivers, filesystems, etc.), a resource manager or other server program can be easily *decoupled from the OS*.

This decoupling lets you safely stop, start, and upgrade resource managers or other key programs *dynamically*, without compromising the availability of the rest of the system.

Consider the following code, where we restart the `inetd` daemon:

```
/* addinet.c */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/netmgr.h>
#include <fcntl.h>
#include "ha/ham.h"

int main(int argc, char *argv[])
{
      int status;
      char *inetdpath;
    ham_entity_t *ehdl;
    ham_condition_t *chdl;
    ham_action_t *ahdl;
    int inetdpid;
    if (argc > 1)
        inetdpath = strdup(argv[1]);
    else
        inetdpath = strdup("/usr/sbin/inetd -D");
    if (argc > 2)
        inetdpid = atoi(argv[2]);
    else
        inetdpid = -1;
    ham_connect(0);
    ehdl = ham_attach("inetd", ND_LOCAL_NODE, inetdpid, inetdpath, 0);
    if (ehdl != NULL)
    {
      chdl = ham_condition(ehdl,CONDDEATH, "death", HREARMAFTERRESTART);
    if (chdl != NULL) {
        ahdl = ham_action_restart(chdl, "restart", inetdpath,
                              HREARMAFTERRESTART);
          if (ahdl == NULL)
              printf("add action failed\n");
        }
        else
            printf("add condition failed\n");
    }
    else
        printf("add entity failed\n");
    ham_disconnect(0);
    exit(0);
}
```

The above example attaches the `inetd` process to a HAM, and then establishes a condition *death* and an action *restart* under it.

If `inetd` isn't a self-attached entity, you need to specify the `-D` option to it, to force `inetd` to daemonize by calling *procmgr_daemon()* instead of by calling *daemon()*. The HAM can see death messages only from self-attached entities, processes that terminate abnormally, and tasks that are running in session 1, and the call to *daemon()* doesn't put the caller into that session.

If `inetd` is a self-attached entity, you don't need to specify the `-D` option because the HAM automatically switches to monitoring the new process that *daemon()* creates.

When `inetd` terminates, the HAM will automatically restart it by running the program specified by `inetdpath`. If `inetd` were already running on the system, we can pass the *pid* of the existing `inetd` into `inetdpid` and it will be attached to directly. Otherwise, the HAM will start and begin to monitor `inetd`.

You could use the same code to monitor, say, `slogger` (by specifying `/usr/sbin/slogger`), `mqueue` (by specifying `/sbin/mqueue`), etc. Just remember to specify the *full path* of the executable with all its required command-line parameters.

Compound restart

Recovery often involves more than restarting a single component. The death of one component might actually require restarting and resetting many other components. We might also have to do some initial cleanup before the dead component is restarted.

A HAM lets you specify a list of actions that will be performed when a given condition is triggered. For example, suppose the entity being monitored is `fs-nfs2`, and there's a set of directories that have been mounted and are currently in use. If `fs-nfs2` were to die, the simple restart of that component won't remount the directories and make them available again! We'd have to restart `fs-nfs2`, and then follow that up with the explicit mounting of the appropriate directories.

Similarly, if `io-pkt*` were to die, it would take down the network drivers and TCP/IP stack (`npm-tcpip.so`) with it. So restarting `io-pkt*` involves also reinitializing the network driver. Also, any other components that use the network connection will also need to be reset (like `inetd`) so that they can reestablish their connections again.

Consider the following example of performing a compound restart mechanism.

```
/* addnfs.c */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/netmgr.h>
#include <fcntl.h>
#include <ha/ham.h>

int main(int argc, char *argv[])
{
      int status;
    ham_entity_t *ehdl;
    ham_condition_t *chdl;
    ham_action_t *ahdl;
    char *fsnfspath;
    int fsnfs2pid;
    if (argc > 1)
        fsnfspath = strdup(argv[1]);
    else
        fsnfspath = strdup("/usr/sbin/fs-nfs2");
    if (argc > 2)
        fsnfs2pid = atoi(argv[2]);
    else
        fsnfs2pid = -1;
    ham_connect(0);
    ehdl = ham_attach("Fs-nfs2", ND_LOCAL_NODE, fsnfs2pid, fsnfspath, 0);
    if (ehdl != NULL)
    {
      chdl = ham_condition(ehdl,CONDDEATH, "Death", HREARMAFTERRESTART);
    if (chdl != NULL) {
        ahdl = ham_action_restart(chdl, "Restart", fsnfspath,
                                  HREARMAFTERRESTART);
        if (ahdl == NULL)
            printf("add action failed\n");
        /* else {
          ahdl = ham_action_waitfor(chdl, "Delay1", NULL, 2000, HREARMAFTERRESTART);

          if (ahdl == NULL)
              printf("add action failed\n");
          ahdl = ham_action_execute(chdl, "MountARMLE-v7",
              "/bin/mount -t nfs 10.12.1.115:/armle-v7 /armle-v7",
              HREARMAFTERRESTART|((fsnfs2pid == -1) ? HACTIONDONOW:0));

          if (ahdl == NULL)
              printf("add action failed\n");

          ahdl = ham_action_waitfor(chdl, "Delay2", NULL, 2000, HREARMAFTERRESTART);

          if (ahdl == NULL)
              printf("add action failed\n");

          ahdl = ham_action_execute(chdl, "MountWeb",
              "/bin/mount -t nfs 10.12.1.115:/web /web",
              HREARMAFTERRESTART|((fsnfs2pid == -1) ? HACTIONDONOW:0));

          if (ahdl == NULL)
              printf("add action failed\n");
        } */
        }
        else
            printf("add condition failed\n");
    }
    else
        printf("add entity failed\n");
    ham_disconnect(0);
    exit(0);
}
```

This example attaches `fs-nfs2` as an entity, and then attaches a series of *execute* and *waitfor* actions to the condition *death*. When `fs-nfs2` dies, HAM will restart it and also remount the remote directories that need to be remounted in sequence. Note that you can specify *delays* as actions and also wait for specific names to appear in the namespace.

Death/condition notification

Fault notification is a crucial part of the availability of a system. Apart from performing recovery per se, we also need to keep track of failures in order to be able to analyze the system at a later point.

For fault notification, you can use standard notification mechanisms such as pulses or signals. Clients specify what pulse/signal with specific values they want for each notification, and a HAM delivers the notifications at the appropriate times.

```
/* regevent.c */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/netmgr.h>
#include <signal.h>
#include <ha/ham.h>

#define PCODEINETDDEATH      _PULSE_CODE_MINAVAIL+1
#define PCODEINETDDETACH     _PULSE_CODE_MINAVAIL+2
#define PCODENFSDELAYED      _PULSE_CODE_MINAVAIL+3
#define PCODEINETDRESTART1   _PULSE_CODE_MINAVAIL+4
#define PCODEINETDRESTART2   _PULSE_CODE_MINAVAIL+5

#define MYSIG SIGRTMIN+1

int fsnfs_value;

/* Signal handler to handle the death notify of fs-nfs2 */
void MySigHandler(int signo, siginfo_t *info, void *extra)
{
  printf("Received signal %d, with code = %d, value %d\n",
        signo, info->si_code, info->si_value.sival_int);
  if (info->si_value.sival_int == fsnfs_value)
    printf("FS-nfs2 died, this is the notify signal\n");
  return;
}

int main(int argc, char *argv[])
{
  int chid, coid, rcvid;
  struct _pulse pulse;
  pid_t pid;
  int status;
  int value;
  ham_entity_t *ehdl;
  ham_condition_t *chdl;
  ham_action_t *ahdl;
  struct sigaction sa;
  int scode;
  int svalue;

  /* we need a channel to receive the pulse notification on */
  chid = ChannelCreate( 0 );

  /* and we need a connection to that channel for the pulse to be
     delivered on */
  coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );

  /* fill in the event structure for a pulse */
  pid = getpid();
  value = 13;
  ham_connect(0);
```

```
/* Assumes there is already an entity by the name "inetd" */
chdl = ham_condition_handle(ND_LOCAL_NODE, "inetd","death",0);
ahdl = ham_action_notify_pulse(chdl, "notifypulsedeath",ND_LOCAL_NODE, pid,
                            chid, PCODEINETDDEATH, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ehdl = ham_entity_handle(ND_LOCAL_NODE, "inetd", 0);
chdl = ham_condition(ehdl, CONDDETACH, "detach", HREARMAFTERRESTART);
ahdl = ham_action_notify_pulse(chdl, "notifypulsedetach",ND_LOCAL_NODE, pid,
                             chid, PCODEINETDDETACH, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ham_entity_handle_free(ehdl);
fsnfs_value = 18; /* value we expect when fs-nfs dies */
scode = 0;
svalue = fsnfs_value;
sa.sa_sigaction = MySigHandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_SIGINFO;
sigaction(MYSIG, &sa, NULL);
/*
 Assumes there is an entity by the name "Fs-nfs2".
 We use "Fs-nfs2" to symbolically represent the entity
 fs-nfs2. Any name can be used to represent the
 entity, but it's best to use a readable and meaningful name.
*/
ehdl = ham_entity_handle(ND_LOCAL_NODE, "Fs-nfs2", 0);

/*
 Add a new condition, which will be an "independent" condition
 this means that notifications/actions inside this condition
 are not affected by "waitfor" delays in other action
 sequence threads
*/
chdl = ham_condition(ehdl,CONDDEATH, "DeathSep",
                  HCONDINDEPENDENT|HREARMAFTERRESTART);
ahdl = ham_action_notify_signal(chdl, "notifysignaldeath",ND_LOCAL_NODE, pid,
                  MYSIG, scode, svalue, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ham_entity_handle_free(ehdl);
chdl = ham_condition_handle(ND_LOCAL_NODE, "Fs-nfs2","Death",0);
/*
 this actions is added to a condition that does not
 have a hcondnowait. Since we are unaware what the condition
 already contains, we might end up getting a delayed notification
 since the action sequence might have "arbitrary" delays, and
 "waits" in it.
*/
ahdl = ham_action_notify_pulse(chdl, "delayednfsdeathpulse", ND_LOCAL_NODE,
           pid, chid, PCODENFSDELAYED, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ehdl = ham_entity_handle(ND_LOCAL_NODE, "inetd", 0);
chdl = ham_condition(ehdl, CONDRESTART, "restart",
                          HREARMAFTERRESTART|HCONDINDEPENDENT);
ahdl = ham_action_notify_pulse(chdl, "notifyrestart_imm", ND_LOCAL_NODE,
                  pid, chid, PCODEINETDRESTART1, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ahdl = ham_action_waitfor(chdl, "delay",NULL,6532, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ahdl = ham_action_notify_pulse(chdl, "notifyrestart_delayed", ND_LOCAL_NODE,
                  pid, chid, PCODEINETDRESTART2, value, HREARMAFTERRESTART);
ham_action_handle_free(ahdl);
ham_condition_handle_free(chdl);
ham_entity_handle_free(ehdl);
while (1) {
  rcvid = MsgReceivePulse( chid, &pulse, sizeof( pulse ), NULL );
  if (rcvid < 0) {
    if (errno != EINTR) {
      exit(-1);
    }
  }
  else {
          switch (pulse.code) {
              case PCODEINETDDEATH:
                    printf("Inetd Death Pulse\n");
                  break;
              case PCODENFSDELAYED:
                    printf("Fs-nfs2 died: this is the possibly delayed pulse\n");
```

```
                      break;
                case PCODEINETDDETACH:
                      printf("Inetd detached, so quitting\n");
                     goto the_end;
                case PCODEINETDRESTART1:
                      printf("Inetd Restart Pulse: Immediate\n");
                    break;
                case PCODEINETDRESTART2:
                      printf("Inetd Restart Pulse: Delayed\n");
                    break;
              }
    }
  }
  /*
   At this point we are no longer waiting for the
   information about inetd, since we know that it
   has exited.
   We will still continue to obtain information about the
   death of fs-nfs2, since we did not remove those actions
   if we exit now, the next time those actions are executed
   they will fail (notifications fail if the receiver does
   exist anymore), and they will automatically get removed and
   cleaned up.
  */
the_end:
  ham_disconnect(0);
  exit(0);
}
```

In the above example a client registers for various different types of notifications relating to significant events concerning `inetd` and `fs-nfs2`. Notifications can be sent immediately or after a certain delay.

The notifications can also be received for each condition *independently* — for the entity's death (`CONDDEATH`), restart (`CONDRESTART`), and detaching (`CONDDETACH`).

The `CONDRESTART` is asserted by a HAM when an entity is successfully restarted.

Heartbeating clients (liveness detection)

Sometimes components become unavailable not because of the occurrence of a specific “bad” event, but because the components become unresponsive by getting stuck somewhere to the extent that the service they provide becomes effectively unavailable.

One example of this is when a process or a collection of processes/threads enters a state of deadlock or starvation, where none or only some of the involved processes can make any useful progress. Such situations are often difficult to pinpoint since they occur quite randomly.

You can have your clients assert “liveness” properties by actively sending heartbeats to a HAM. When a process deadlocks (or starves) and makes no progress, it will no longer heartbeat, and the HAM will automatically detect this condition and take corrective action.

The corrective action can range from simply terminating the offending application to restarting it and also delivering notifications about its state to other components that depend on the safe and correct functioning of this component. If necessary, a HAM can restart those other components as well.

We can demonstrate this condition by showing a simple process that has two threads that use mutual-exclusion locks incorrectly (by a design flaw), which causes them on occasion to enter a state of deadlock — each of the threads holds a resource that the other wants.

Essentially, each thread runs through a segment of code that involves the use of two mutexes.

```
Thread 1                             Thread 2

...                                  ...
while true                           while true
  do                                 do
    obtain lock a                      obtain lock b
    (compute section1)                   (compute section1)
    obtain lock b                        obtain lock a
        (compute section2)                 (compute section2)
    release lock b                       release lock a
    release lock a                     release lock b
done                                 done
...                                  ...
```

The code segments for each thread are shown below. The only difference between the two is the order in which the locks are obtained. The two threads deadlock upon execution, quite randomly; the exact moment of deadlock is related to the lengths of the “compute sections” of the two threads.

```
/* mutexdeadlock.c */

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

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <pthread.h>
#include <process.h>
#include <sys/neutrino.h>
#include <sys/procfs.h>
#include <sys/procmgr.h>
#include <ha/ham.h>

pthread_mutex_t mutex_a = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_b = PTHREAD_MUTEX_INITIALIZER;

FILE *logfile;
pthread_t  threadID;
int doheartbeat=0;

#define COMPUTE_DELAY 100

void *func1(void *arg)
{
    int id;
    /* obtain the two locks in the order
       a -> b
       perform some computation and then
       release the locks ...
       do this continuously
    */

    id = pthread_self();
    while (1) {
        delay(85); /* delay to let the other one go */
        if (doheartbeat)
            ham_heartbeat();
        pthread_mutex_lock(&mutex_a);
        fprintf(logfile, "Thread 1: Obtained lock a\n");
        fprintf(logfile, "Thread 1: Waiting for lock b\n");
        pthread_mutex_lock(&mutex_b);
        fprintf(logfile, "Thread 1: Obtained lock b\n");
        fprintf(logfile, "Thread 1: Performing computation\n");
        delay(rand()%COMPUTE_DELAY+5); /* delay for computation */
        fprintf(logfile, "Thread 1: Unlocking lock b\n");
        pthread_mutex_unlock(&mutex_b);
        fprintf(logfile, "Thread 1: Unlocking lock a\n");
        pthread_mutex_unlock(&mutex_a);
    }
    return(NULL);
}

void *func2(void *arg)
{

    int id;
    /* obtain the two locks in the order
       b -> a
       perform some computation and then
       release the locks ...
       do this continuously
    */

    id = pthread_self();
    while (1) {
        delay(25);
        if (doheartbeat)
            ham_heartbeat();
        pthread_mutex_lock(&mutex_b);
        fprintf(logfile, "\tThread 2: Obtained lock b\n");
        fprintf(logfile, "\tThread 2: Waiting for lock a\n");
        pthread_mutex_lock(&mutex_a);
        fprintf(logfile, "\tThread 2: Obtained lock a\n");
        fprintf(logfile, "\tThread 2: Performing computation\n");
        delay(rand()%COMPUTE_DELAY+5); /* delay for computation */
        fprintf(logfile, "\tThread 2: Unlocking lock a\n");
        pthread_mutex_unlock(&mutex_a);
        fprintf(logfile, "\tThread 2: Unlocking lock b\n");
        pthread_mutex_unlock(&mutex_b);
    }
    return(NULL);
}
```

```
int main(int argc, char *argv[])
{
    pthread_attr_t attrib;
    struct sched_param param;
    ham_entity_t *ehdl;
    ham_condition_t *chdl;
    ham_action_t *ahdl;
    int i=0;
    char c;

    logfile = stderr;
    while ((c = getopt(argc, argv, "f:l" )) != -1 ) {
        switch(c) {
            case 'f': /* log file */
                logfile = fopen(optarg, "w");
                break;
            case 'l': /* do liveness heartbeating */
                if (access("/proc/ham",F_OK) == 0)
                    doheartbeat=1;
                break;
         }
    }

    setbuf(logfile, NULL);
    srand(time(NULL));
    fprintf(logfile, "Creating separate competing compute thread\n");

    pthread_attr_init (&attrib);
    pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy (&attrib, SCHED_RR);
    param.sched_priority = getprio (0);
    pthread_attr_setschedparam (&attrib, &param);

    if (doheartbeat) {
        /* attach to ham */
        ehdl = ham_attach_self("mutex-deadlock",1000000000UL,5 ,5, 0);
        chdl = ham_condition(ehdl, CONDHBEATMISSEDHIGH, "heartbeat-missed-high", 0);
        ahdl = ham_action_execute(chdl, "terminate",
                              "/proc/boot/mutex-deadlock-heartbeat.sh", 0);
    }
    /* create competitor thread */
    pthread_create (&threadID, &attrib, func1, NULL);
    pthread_detach(threadID);

    func2(NULL);

    exit(0);
}
```

Upon execution, what we see is:

1. Starting two-threaded process.

   The threads will execute as described earlier, but will eventually deadlock. We'll wait for a reasonable amount of time (a few seconds) until they do end in deadlock. The threads write out a simple execution log into `/dev/shmem/mutex-deadlock.log`.

2. Waiting for them to deadlock.

   Here's the current state of the threads in process 73746:

```
     pid tid name               prio STATE       Blocked
   73746   1 oot/mutex-deadlock  10r MUTEX       73746-02 #-21474
   73746   2 oot/mutex-deadlock  10r MUTEX       73746-01 #-21474
```

   And here's the tail from the threads' log file:

```
Thread 2: Obtained lock b
Thread 2: Waiting for lock a
Thread 2: Obtained lock a
Thread 2: Performing computation
Thread 2: Unlocking lock a
Thread 2: Unlocking lock b
Thread 2: Obtained lock b
Thread 2: Waiting for lock a
Thread 1: Obtained lock a
Thread 1: Waiting for lock b
```

3. Extracting core current process information:

```
/tmp/mutex-deadlock.core:
 processor=ARM num_cpus=2
  cpu 1 cpu=602370 name=604e speed=299
  flags=0xc0000001 FPU MMU EAR
  cpu 2 cpu=602370 name=604e speed=299
   flags=0xc0000001 FPU MMU EAR
 cyc/sec=16666666 tod_adj=999522656000000000 nsec=5190771360840 inc=999960

 boot=999522656 epoch=1970 intr=-2147483648
 rate=600000024 scale=-16 load=16666
   MACHINE="mtx604-smp" HOSTNAME="localhost"
 hwflags=0x000004
 pretend_cpu=0 init_msr=36866
 pid=73746 parent=49169 child=0 pgrp=73746 sid=1
 flags=0x000300 umask=0 base_addr=0x48040000 init_stack=0x4803fa20
 ruid=0 euid=0 suid=0  rgid=0 egid=0 sgid=0
 ign=0000000006801000 queue=ff00000000000000 pending=0000000000000000
 fds=4 threads=2 timers=0 chans=1
 thread 1 REQUESTED
  ip=0xfe32f838 sp=0x4803f920 stkbase=0x47fbf000 stksize=528384
  state=MUTEX flags=0 last_cpu=1 timeout=00000000
  pri=10 realpri=10 policy=RR
 thread 2
  ip=0xfe32f838 sp=0x47fbef80 stkbase=0x47f9e000 stksize=135168
  state=MUTEX flags=4020000 last_cpu=2 timeout=00000000
  pri=10 realpri=10 policy=RR
```

The processes are deadlocked, with each process holding one lock and waiting for the other.

The process is made to heartbeat

Now consider the case where the client can be made to heartbeat so that a HAM will automatically detect when it's unresponsive and will terminate it.

```
Thread 1                            Thread 2

...                                 ...
while true                          while true
do                                  do
  obtain lock a                       obtain lock b
    (compute section1)                  (compute section1)
    obtain lock b                       obtain lock a
      send heartbeat                      send heartbeat
      (compute section2)                  (compute section2)
    release lock b                      release lock a
  release lock a                      release lock b
done                                done
...                                 ...
```

Here the process is expected to send heartbeats to a HAM. By placing the heartbeat call within the inside loop, the deadlock condition is trapped. The HAM notices that the heartbeats have stopped and can then perform recovery.

Let's look at what happens now:

1. Starting two-threaded process.

   The threads will execute as described earlier, but will eventually deadlock. We'll wait for a reasonable amount of time (a few seconds) until they do end in deadlock.

The threads write out a simple execution log into `/dev/shmem/mutex-deadlock-heartbeat.log`. The HAM detects that the threads have stopped heartbeating and terminates the process, after saving its state for postmortem analysis.

2. Waiting for them to deadlock.

   Here's the current state of the threads in process 462866 and the state of mutex-deadlock when it missed heartbeats:

```
     pid tid name               prio STATE       Blocked
  462866   1 oot/mutex-deadlock  10r MUTEX       462866-03 #-2147
  462866   2 oot/mutex-deadlock  63r RECEIVE     1
  462866   3 oot/mutex-deadlock  10r MUTEX       462866-01 #-2147


    Entity state from HAM

Path            : mutex-deadlock
Entity Pid      : 462866
Num conditions  : 1
Condition type  : ATTACHEDSELF
Stats:
HeartBeat Period: 1000000000
HB Low Mark     : 5
HB High Mark    : 5
Last Heartbeat  : 2001/09/03 14:40:41:406575120
HeartBeat State : MISSEDHIGH
Created         : 2001/09/03 14:40:40:391615720
Num Restarts    : 0
```

   And here's the tail from the threads' log file:

```
Thread 2: Obtained lock b
Thread 2: Waiting for lock a
Thread 2: Obtained lock a
Thread 2: Performing computation
Thread 2: Unlocking lock a
Thread 2: Unlocking lock b
Thread 2: Obtained lock b
Thread 2: Waiting for lock a
Thread 1: Obtained lock a
Thread 1: Waiting for lock b
```

3. Extracting core current process information:

```
/tmp/mutex-deadlock.core:
 processor=ARM num_cpus=2
  cpu 1 cpu=602370 name=604e speed=299
   flags=0xc0000001 FPU MMU EAR
  cpu 2 cpu=602370 name=604e speed=299
   flags=0xc0000001 FPU MMU EAR
 cyc/sec=16666666 tod_adj=999522656000000000 nsec=5390696363520 inc=999960

 boot=999522656 epoch=1970 intr=-2147483648
 rate=600000024 scale=-16 load=16666
   MACHINE="mtx604-smp" HOSTNAME="localhost"
 hwflags=0x000004
 pretend_cpu=0 init_msr=36866
 pid=462866 parent=434193 child=0 pgrp=462866 sid=1
 flags=0x000300 umask=0 base_addr=0x48040000 init_stack=0x4803f9f0
 ruid=0 euid=0 suid=0  rgid=0 egid=0 sgid=0
 ign=0000000006801000 queue=ff00000000000000 pending=0000000000000000
 fds=5 threads=3 timers=1 chans=4
 thread 1 REQUESTED
  ip=0xfe32f838 sp=0x4803f8f0 stkbase=0x47fbf000 stksize=528384
  state=MUTEX flags=0 last_cpu=2 timeout=00000000
  pri=10 realpri=10 policy=RR
 thread 2
  ip=0xfe32f1a8 sp=0x47fbef50 stkbase=0x47f9e000 stksize=135168
  state=RECEIVE flags=4000000 last_cpu=2 timeout=00000000
  pri=63 realpri=63 policy=RR
  blocked_chid=1
 thread 3
```

```
ip=0xfe32f838 sp=0x47f9df80 stkbase=0x47f7d000 stksize=135168
state=MUTEX flags=4020000 last_cpu=1 timeout=00000000
pri=10 realpri=10 policy=RR
```

Process starvation

We can demonstrate this condition by showing a simple process containing two threads that use mutual exclusion locks to manage a critical section. Thread 1 runs at a high priority, while Thread 2 runs at a lower priority. Essentially, each thread runs through a segment of code that looks like this:

```
Thread1                              Thread 2

...                                  ...
(Run at high priority)               (Run at low priority)
while true                           while true
do                                   do
    obtain lock a                        obtain lock a
        (compute section1)                   (compute section1)
    release lock a                       release lock a
done                                 done
...                                  ...
```

The code segments for each thread is shown below; the only difference being the priorities of the two threads. Upon execution, Thread 2 eventually starves.

```
/* mutexstarvation.c */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <pthread.h>
#include <process.h>
#include <sys/neutrino.h>
#include <sys/procfs.h>
#include <sys/procmgr.h>
#include <ha/ham.h>

pthread_mutex_t mutex_a = PTHREAD_MUTEX_INITIALIZER;

FILE *logfile;
int doheartbeat=0;

#define COMPUTE_DELAY 900

void *func1(void *arg)
{
    int id;

    id = pthread_self();
    while (1) {
        pthread_mutex_lock(&mutex_a);
        fprintf(logfile, "Thread 1: Locking lock a\n");
        delay(rand()%COMPUTE_DELAY+50); /* delay for computation */
        fprintf(logfile, "Thread 1: Unlocking lock a\n");
        pthread_mutex_unlock(&mutex_a);
    }
    return(NULL);
}

void *func2(void *arg)
{

    int id;

    id = pthread_self();
    while (1) {
        pthread_mutex_lock(&mutex_a);
        fprintf(logfile, "\tThread 2: Locking lock a\n");
        if (doheartbeat)
            ham_heartbeat();
        delay(rand()%COMPUTE_DELAY+50); /* delay for computation */
        fprintf(logfile, "\tThread 2: Unlocking lock a\n");
        pthread_mutex_unlock(&mutex_a);
    }
    return(NULL);
}

int main(int argc, char *argv[])
{
    pthread_attr_t attrib;
    struct sched_param param;
    ham_entity_t *ehdl;
    ham_condition_t *chdl;
    ham_action_t *ahdl;
    int i=0;
    char c;
    pthread_attr_t attrib2;
    struct sched_param param2;
    pthread_t  threadID;
    pthread_t  threadID2;

    logfile = stderr;
    while ((c = getopt(argc, argv, "f:l" )) != -1 ) {
        switch(c) {
            case 'f': /* log file */
                logfile = fopen(optarg, "w");
                break;
            case 'l': /* do liveness heartbeating */
                if (access("/proc/ham",F_OK) == 0)
                    doheartbeat=1;
                break;
         }
    }

    setbuf(logfile, NULL);
    srand(time(NULL));
    fprintf(logfile, "Creating separate competing compute thread\n");

    if (doheartbeat) {
        /* attach to ham */
        ehdl = ham_attach_self("mutex-starvation",1000000000UL, 5, 5, 0);
        chdl = ham_condition(ehdl, CONDHBEATMISSEDHIGH, "heartbeat-missed-high", 0);
        ahdl = ham_action_execute(chdl, "terminate",
                               "/proc/boot/mutex-starvation-heartbeat.sh", 0);
    }
    /* create competitor thread */
    pthread_attr_init (&attrib2);
    pthread_attr_setinheritsched (&attrib2, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy (&attrib2, SCHED_RR);
    param2.sched_priority = sched_get_priority_min(SCHED_RR);
    pthread_attr_setschedparam (&attrib2, &param2);

    pthread_create (&threadID2, &attrib2, func2, NULL);

    delay(3000); /* let the other thread go on for a while... */

    pthread_attr_init (&attrib);
    pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy (&attrib, SCHED_RR);
    param.sched_priority = sched_get_priority_max(SCHED_RR);
    pthread_attr_setschedparam (&attrib, &param);

    pthread_create (&threadID, &attrib, func1, NULL);

    pthread_join(threadID, NULL);
    pthread_join(threadID2, NULL);
    exit(0);
}
```

Upon execution, here's what we see:

1. Starting two-threaded process.

The threads will execute as described earlier, but eventually Thread 2 will starve. We'll wait for a reasonable amount of time (some seconds) until Thread 2 ends up starving. The threads write out a simple execution log into `/dev/shmem/mutex-starvation.log`.

2. Waiting for them to run for a while.

   Here's the current state of the threads in process 622610:

```
     pid tid name               prio STATE       Blocked
  622610   1 t/mutex-starvation  10r JOIN        3
  622610   2 t/mutex-starvation   1r MUTEX       622610-03 #-2147
  622610   3 t/mutex-starvation  63r NANOSLEEP
```

   And here's the tail from the threads' log file:

```
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
```

3. Extracting core current process information:

```
/tmp/mutex-starvation.core:
 processor=ARM num_cpus=2
  cpu 1 cpu=602370 name=604e speed=299
   flags=0xc0000001 FPU MMU EAR
  cpu 2 cpu=602370 name=604e speed=299
   flags=0xc0000001 FPU MMU EAR
 cyc/sec=16666666 tod_adj=999522656000000000 nsec=5561011550640 inc=999960

 boot=999522656 epoch=1970 intr=-2147483648
 rate=600000024 scale=-16 load=16666
   MACHINE="mtx604-smp" HOSTNAME="localhost"
 hwflags=0x000004
 pretend_cpu=0 init_msr=36866
 pid=622610 parent=598033 child=0 pgrp=622610 sid=1
 flags=0x000300 umask=0 base_addr=0x48040000 init_stack=0x4803fa10
 ruid=0 euid=0 suid=0  rgid=0 egid=0 sgid=0
 ign=0000000006801000 queue=ff00000000000000 pending=0000000000000000
 fds=4 threads=3 timers=0 chans=1
 thread 1 REQUESTED
  ip=0xfe32f8c8 sp=0x4803f8a0 stkbase=0x47fbf000 stksize=528384
  state=JOIN flags=0 last_cpu=1 timeout=00000000
  pri=10 realpri=10 policy=RR
 thread 2
  ip=0xfe32f838 sp=0x47fbef80 stkbase=0x47f9e000 stksize=135168
  state=MUTEX flags=4000000 last_cpu=2 timeout=00000000
  pri=1 realpri=1 policy=RR
 thread 3
  ip=0xfe32f9a0 sp=0x47f9df20 stkbase=0x47f7d000 stksize=135168
  state=NANOSLEEP flags=4000000 last_cpu=2 timeout=0x1001000
  pri=63 realpri=63 policy=RR
```

Thread 2 is made to heartbeat

Now consider the case where Thread 2 is made to heartbeat. A HAM will automatically detect when the thread is unresponsive and can terminate it and/or perform recovery.

```
Thread 1                            Thread 2
```

```
...                                     ...
(Run at high priority)                  (Run at low priority)
while true                              while true
do                                      do
    obtain lock a                           obtain lock a
                                                send heartbeat
        (compute section1)                      (compute section1)
    release lock a                          release lock a
done                                    done
...                                     ...
```

Here Thread 2 is expected to send heartbeats to a HAM. By placing the heartbeat call within the inside loop, the HAM detects when Thread 2 begins to starve.

The threads will execute as described earlier, but eventually Thread 2 will starve. We'll wait for a reasonable amount of time (some seconds) until it does. The threads write out a simple execution log into `/dev/shmem/mutex-starvation-heartbeat.log`. The HAM detects that the thread has stopped heartbeating and terminates the process, after saving its state for postmortem analysis.

Let's look at what happens:

1. Waiting for some time.

Here's the current state of the threads in process 753682 and the state of mutex-starvation when it missed heartbeats:

```
     pid tid name                  prio STATE       Blocked
  753682   1 t/mutex-starvation  10r JOIN        4
  753682   2 t/mutex-starvation  63r RECEIVE     1
  753682   3 t/mutex-starvation   1r MUTEX       753682-04 #-2147
  753682   4 t/mutex-starvation  63r NANOSLEEP


    Entity state from HAM

Path            : mutex-starvation
Entity Pid      : 753682
Num conditions  : 1
Condition type  : ATTACHEDSELF
Stats:
HeartBeat Period: 1000000000
HB Low Mark     : 5
HB High Mark    : 5
Last Heartbeat  : 2001/09/03 14:44:37:796119160
HeartBeat State : MISSEDHIGH
Created         : 2001/09/03 14:44:34:780239800
Num Restarts    : 0
```

And here's the tail from the threads' log file:

```
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
Thread 1: Unlocking lock a
Thread 1: Locking lock a
```

2. Extracting core current process information:

```
/tmp/mutex-starvation.core:
 processor=ARM num_cpus=2
  cpu 1 cpu=602370 name=604e speed=299
   flags=0xc0000001 FPU MMU EAR
  cpu 2 cpu=602370 name=604e speed=299
```

```
    flags=0xc0000001 FPU MMU EAR
  cyc/sec=16666666 tod_adj=999522656000000000 nsec=5627098907040 inc=999960

  boot=999522656 epoch=1970 intr=-2147483648
  rate=600000024 scale=-16 load=16666
    MACHINE="mtx604-smp" HOSTNAME="localhost"
  hwflags=0x000004
  pretend_cpu=0 init_msr=36866
 pid=753682 parent=729105 child=0 pgrp=753682 sid=1
  flags=0x000300 umask=0 base_addr=0x48040000 init_stack=0x4803f9f0
  ruid=0 euid=0 suid=0  rgid=0 egid=0 sgid=0
  ign=0000000006801000 queue=ff00000000000000 pending=0000000000000000
  fds=5 threads=4 timers=1 chans=4
  thread 1 REQUESTED
   ip=0xfe32f8c8 sp=0x4803f880 stkbase=0x47fbf000 stksize=528384
   state=JOIN flags=0 last_cpu=2 timeout=00000000
   pri=10 realpri=10 policy=RR
  thread 2
   ip=0xfe32f1a8 sp=0x47fbef50 stkbase=0x47f9e000 stksize=135168
   state=RECEIVE flags=4000000 last_cpu=2 timeout=00000000
   pri=63 realpri=63 policy=RR
   blocked_chid=1
  thread 3
   ip=0xfe32f838 sp=0x47f9df80 stkbase=0x47f7d000 stksize=135168
   state=MUTEX flags=4000000 last_cpu=2 timeout=00000000
   pri=1 realpri=1 policy=RR
  thread 4
   ip=0xfe32f9a0 sp=0x47f7cf20 stkbase=0x47f5c000 stksize=135168
   state=NANOSLEEP flags=4000000 last_cpu=1 timeout=0x1001000
   pri=63 realpri=63 policy=RR
```

Glossary

action

A specific task the HAM will perform under certain associated *conditions*. Examples of actions include executing an external process, restarting a process that has died, sending a signal or pulse notification, etc.

availability

The ability of a system to provide its intended service without interruption for extended periods of time.

clustering

A method of distributing processing among several computers in order to reduce the number of *SPOFs*. QNX Neutrino native networking offers transparent network-wide processing, which facilitates building clustered HA applications.

condition

An event that will trigger certain *actions* for the HAM to perform. Examples of conditions include the death of entity, a missed heartbeat, etc.

entity

A process that the HAM will monitor. Entities can explicitly ask to be monitored (i.e., as *self-attached* entities), or they may be monitored without ever realizing it.

five nines

The celebrated *availability* metric that refers to a system's ability to remain up and running 99.999% of the time per year.

Guardian

The HAM's "clone", a stand-in process that the HAM creates to ensure uninterrupted HA management within the QNX Neutrino environment.

HAM

High Availability Manager.

heartbeat

A "wellness" or "liveness" notification sent at specific intervals by a client to the HAM.

hot swap

The ability to remove or insert a component in a live system.

MMU

Memory Management Unit. A device on many CPUs that alerts the OS if a process tries to access memory that's been allocated to another process.

MTTF

Mean Time To Failure. This is the average length of time that the system will remain in service before failing. You want this to be as long as possible.

MTTR

Mean Time To Repair. This is the amount of time it takes for the system to resume operation after any component fails or is upgraded. You want this to be as small as possible.

SPOF

Single point of failure. Any particular “weak link” in a system would be considered a SPOF, because its demise would put the entire system at risk.

watchdog

A trusted piece of hardware whose main purpose is to trigger code that will check the sanity of the system. There are software watchdogs as well; the HAM may be considered a “smart watchdog.”

Index

"five nines" metric 11

A

action 27, 69, 72, 76, 78, 80, 83, 86, 94, 102, 104, 105, 107
  adding a notify pulse action to an action on failure 80
  adding a notify signal action to an action on failure 83
  adding a restart 104
  control 69
  defined 27
  executing a waitfor action on action fail 86
  executing an external command 72
  executing an external command on action fail 76
  inserting delays into a sequence of actions 107
  log activity 94
  logging messages on action fail 78
  removing from a condition 102
  restart can be associated only with death conditions 104
  restarting a program 105
action_fail 28
  defined 28

C

checkpoint 63
close() 167
code example 190, 192, 194, 197, 203
  compound restart 192
  death/condition notification 194
  heartbeating clients (liveness detection) 197
  process starvation 203
  simple restart 190
CONDABNORMALDEATH 39
CONDANY 40
CONDATTACH 39
CONDDEATH 39, 117
CONDDETACH 39, 118
CONDHBEATMISSEDHIGH 39, 118
CONDHBEATMISSEDLOW 40, 118
condition 25, 39, 41, 105, 117, 125, 127, 129, 131
  associated with a condition raise condition 127
  defined 25
  flags 41
  freeing 125
  removing 129
  restart 105
  setting up 117
  triggered on state change 131
  types of 39, 117
CONDRESTART 40, 118
ConnectAttach() 170, 185
ConnectDetach() 172
connection 134, 142, 164, 167, 170, 172, 174, 176, 178, 180, 183, 185, 187
  attaching a recovery function 164

connection *(continued)*
  attaching to the HA lib 170, 174, 183
  closing 142, 167, 172
  creating 170, 174
  detaching 167, 172, 176, 180
  duplicating 178
  opening 183
  reopening during recovery 185, 187
  setting up 134
creat() 174

D

daemon() 25, 32
deadlock 197
death 104
  and restart actions 104
dumper 25
dup() 178

E

entity 24, 25, 88, 110, 114, 117, 121, 123, 136, 138, 140, 150, 153
  attaching 110
  control 121, 150
  defined 24
  detaching 136, 138, 140
  externally attached 25
  getting a handle to an 153
  global 25, 88, 123, 153
    getting a handle to a 88, 123, 153
  handle 117
  self-attached 24, 114, 136, 140
externally attached entity 25

F

failure 14
  designing for in order to recover 14
fclose() 180
file stream 180, 181
  attaching to the HA lib 181
  detaching a connection associated with a 180
fopen() 181

G

global entity 25, 88, 123, 153
  getting a handle to a 88, 123, 153
Guardian (HAM "stand-in") 23, 31, 32, 33

H

HA 15
    microkernel architecture inherently suited for 15
ha_attach() 164, 167, 180, 183
ha_close() 167
ha_ConnectAttach_r() 165
ha_ConnectAttach() 165, 170
ha_ConnectDetach() 172
ha_creat() 165, 174
ha_creat64() 165
ha_detach() 176, 180
ha_dup() 178
ha_fclose() 180
ha_fopen() 165, 181
ha_open() 165, 183
ha_ReConnectAttach() 164, 185
ha_reopen() 164, 187
HADDFLAGS 69, 150
HAM 13, 15, 24, 50
    as self-monitoring manager 15
    hierarchy 24
    multistage recovery 13
    starting 50
    stopping 50
ham (utility) 50
ham_action_control() 69
ham_action_execute() 72, 105
ham_action_fail_execute() 76
ham_action_fail_log() 78
ham_action_fail_notify_pulse_node() 80
ham_action_fail_notify_pulse() 80
ham_action_fail_notify_signal_node() 83
ham_action_fail_notify_signal() 83
ham_action_fail_waitfor() 86
ham_action_handle_free() 90
ham_action_handle() 88
ham_action_heartbeat_healthy() 92
ham_action_log() 94
ham_action_notify_pulse_node() 96
ham_action_notify_pulse() 96
ham_action_notify_signal_node() 99
ham_action_notify_signal() 99
ham_action_remove() 102
ham_action_restart() 104
ham_action_waitfor() 107
ham_attach_node() 110
ham_attach_self() 114
ham_attach() 110
ham_condition_handle_free() 125
ham_condition_handle() 123
ham_condition_raise() 127
ham_condition_remove() 129
ham_condition_state() 131
ham_condition() 117
ham_connect_nd() 134
ham_connect_node() 134
ham_connect() 134
ham_connect*() 134
    reference counts and 134
ham_detach_name_node() 138
ham_detach_name() 138
ham_detach_self() 140
ham_detach() 136
ham_disconnect_nd() 142
ham_disconnect_node() 142
ham_disconnect() 142
    reference counts and 142
ham_entity_condition_raise() 146
ham_entity_condition_state() 148
ham_entity_control() 121, 150
ham_entity_handle_free() 155
ham_entity_handle() 153
ham_entity() 144
ham_heartbeat() 157
ham_stop() 50
ham_verbose() 52, 161
hamctrl (utility) 50, 52
handle 88, 89, 90, 123, 124, 153, 154, 155
    entity 153
    freeing 90, 155
    functions that expect 88
    getting 88
    getting a condition 123
    subsequent use of after a call to ham_action_handle() 89
    subsequent use of after a call to ham_condition_handle() 124
    subsequent use of after a call to ham_entity_handle() 154
HAREPLACERECOVERYFN 165
HCONDINDEPENDENT 118
HCONDNOWAIT 118
HDISABLE 69, 150
heartbeat 92, 115, 157, 197, 200, 206
    as a way to detect deadlock 200
    as a way to detect starvation 206
    clients assert "liveness" by sending 197
    possible delay in receiving 157
    resetting the state of 92
    setting interval for 115
    transmitting 157
HENABLE 69, 150
HGETFLAGS 69, 150
HREARMAFTERRESTART 92, 97, 100, 105, 107, 119
    caveat regarding 100
HRECURSE 151
HREMOVEFLAGS 69, 150
HSETFLAGS 69, 150

I

inetd 190
    restarting 190

M

message passing 21
microkernel 20
MMU 13
MTTR 20

O

open() 135, 183, 187

P

placeholders 144
    for entity objects 144
POSIX process model 20
postmortem analysis 13, 201, 206
procmgr_daemon() 25, 32
pulse 96
    setting up notification of 96

Q

QNX Neutrino 13, 20
    key factors for intrinsic HA 20
    microkernel architecture inherently reduces SPOFs 13

R

recovery 164
    functions 164
        attaching to a connection 164
        defined in 164
restart 104, 105
    action 104
    condition 105

S

self-attached entity 24, 114, 136, 140
session 1 25, 110, 114
signal 99
    setting up notification of 99
software faults 12, 13, 18
    detecting 18
    isolating 13
    main cause of system failure 12
    traditional ways to handle 18
SPOF 13
starting a HAM 50
starvation 205, 206
    condition resulting from mutex problem 205
    detected by HAM 206
stopping a HAM 50

T

Technical support 10
Typographical conventions 8

V

VERBOSE_GET 161
VERBOSE_SET 161
VERBOSE_SET_DECR 161
VERBOSE_SET_INCR 161
verbosity, modifying 52, 161

W

watchdog 18, 23