QNX® Neutrino® RTOS
System Architecture

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: May 02, 2019

Contents

About This Guide

The *System Architecture* guide accompanies the QNX Neutrino RTOS and is intended for both application developers and end-users.

This guide describes the philosophy of QNX Neutrino and the architecture used to robustly implement the OS. It covers message-passing services, followed by the details of the microkernel, the process manager, resource managers, and other aspects of the OS.

The following table may help you find information quickly:

To find out about:	Go to:
OS design goals; message-passing IPC	*The Philosophy of the QNX Neutrino RTOS*
System services	*The QNX Neutrino Microkernel*
Sharing information between processes	*Interprocess Communication (IPC)*
System event monitoring	*The Instrumented Microkernel*
Working on a system with more than one processor	*Multicore Processing*
Memory management, pathname management, etc.	*Process Manager*
Shared objects	*Dynamic Linking*
Device drivers	*Resource Managers*
Image, RAM, Power-Safe, DOS, Flash, NFS, CIFS, Ext2, and other filesystems	*Filesystems*
Persistent Publish/Subscribe (PPS)	*PPS*
Serial and parallel devices	*Character I/O*
Network subsystem	*Networking Architecture*
Native QNX Neutrino networking	*Native Networking (Qnet)*
TCP/IP implementation	*TCP/IP Networking*
Fault recovery	*High Availability*
Sharing resources among competing processes	*Adaptive Partitioning*
An overview of hard and soft real time	*What is Real Time and Why Do I Need It?*
Terms used in QNX Neutrino documentation	*Glossary*

For information about programming, see *Getting Started with QNX Neutrino* and the QNX Neutrino *Programmer's 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`
Constants	NULL
Data types	`unsigned short`
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.

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

DANGER: 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
The Philosophy of the QNX Neutrino RTOS

The primary goal of the QNX Neutrino RTOS is to deliver the open systems POSIX API in a robust, scalable form suitable for a wide range of systems—from tiny, resource-constrained embedded systems to high-end distributed computing environments. The OS supports several processor families, including x86 and ARM.

For mission-critical applications, a robust architecture is also fundamental, so the OS makes flexible and complete use of MMU hardware.

Of course, simply setting out these goals doesn't guarantee results. We invite you to read through this *System Architecture* guide to get a feel for our implementation approach and the design trade-offs chosen to achieve these goals. When you reach the end of this guide, we think you'll agree that QNX Neutrino is the first OS product of its kind to truly deliver open systems standards, wide scalability, and high reliability.

An embeddable POSIX OS?

According to a prevailing myth, if you scratch a POSIX operating system, you'll find UNIX beneath the surface! A POSIX OS is therefore too large and unsuitable for embedded systems.

The fact, however, is that POSIX is *not* UNIX. Although the POSIX standards are rooted in existing UNIX practice, the POSIX working groups explicitly defined the standards in terms of “interface, *not* implementation.”

Thanks to the precise specification within the standards, as well as the availability of POSIX test suites, nontraditional OS architectures can provide a POSIX API without adopting the traditional UNIX kernel. Compare any two POSIX systems and they'll *look* very much alike—they'll have many of the same functions, utilities, etc. But when it comes to performance or reliability, they may be as different as night and day. Architecture makes the difference.

Despite its decidedly non-UNIX architecture, QNX Neutrino implements the standard POSIX API. By adopting a microkernel architecture, the OS delivers this API in a form easily scaled down for realtime embedded systems or incrementally scaled up, as required.

Product scaling

Since you can readily scale a microkernel OS simply by including or omitting the particular processes that provide the functionality required, you can use a single microkernel OS for a much wider range of purposes than you can a realtime executive.

Product development often takes the form of creating a “product line,” with successive models providing greater functionality. Rather than be forced to change operating systems for each version of the product, developers using a microkernel OS can easily scale the system as needed—by adding filesystems, networking, graphical user interfaces, and other technologies.

Some of the advantages to this scalable approach include:

- portable application code (between product-line members)
- common tools used to develop the entire product line
- portable skill sets of development staff
- reduced time-to-market

Why POSIX for embedded systems?

A common problem with realtime application development is that each realtime OS tends to come equipped with its own proprietary API. In the absence of industry standards, this isn't an unusual state for a competitive marketplace to evolve into, since surveys of the realtime marketplace regularly show heavy use of in-house proprietary operating systems. POSIX represents a chance to unify this marketplace.

Among the many POSIX standards, those of most interest to embedded systems developers are:

- *1003.1*—defines the API for process management, device I/O, filesystem I/O, and basic IPC. This encompasses what might be described as the base functionality of a UNIX OS, serving as a useful standard for many applications. From a C-language programming perspective, ANSI X3J11 C is assumed as a starting point, and then the various aspects of managing processes, files, and tty devices are detailed beyond what ANSI C specifies.
- *Realtime Extensions*—defines a set of realtime extensions to the base 1003.1 standard. These extensions consist of semaphores, prioritized process scheduling, realtime extensions to signals, high-resolution timer control, enhanced IPC primitives, synchronous and asynchronous I/O, and a recommendation for realtime contiguous file support.
- *Threads*—further extends the POSIX environment to include the creation and management of multiple threads of execution within a given address space.
- *Additional Realtime Extensions*—defines further extensions to the realtime standard. Facilities such as attaching interrupt handlers are described.
- *Application Environment Profiles*—defines several AEPs (*Realtime AEP*, *Embedded Systems AEP*, etc.) of the POSIX environment to suit different embedded capability sets. These profiles represent embedded OSs with/without filesystems and other capabilities.

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

Apart from any “bandwagon” motive for adopting industry standards, there are several specific advantages to applying the POSIX standard to the embedded realtime marketplace:

Multiple OS sources

Hardware manufacturers are loath to choose a single-sourced hardware component because of the risks implied if that source discontinues production. For the same reason, manufacturers shouldn't be tied to a single-sourced, proprietary OS simply because their application source code isn't portable to other OSs.

By building applications to the POSIX standards, developers can use OSs from multiple vendors. Application source code can be readily ported from platform to platform and from OS to OS, provided that developers avoid using OS-specific extensions.

Portability of development staff

Using a common API for embedded development, programmers experienced with one realtime OS can directly apply their skill sets to other projects involving other processors and operating systems. In addition, programmers with UNIX or POSIX experience can easily work on embedded realtime systems, since the nonrealtime portion of the realtime OS's API is already familiar territory.

Development environment

Even in a cross-hosted development environment, the API remains essentially the same as on the embedded system. Regardless of the particular host (Linux, Windows,...) or the target (x86, ARM), the programmer doesn't need to worry about platform-specific endian, alignment, or I/O issues.

Why QNX Neutrino for embedded systems?

The main responsibility of an operating system is to manage a computer's resources. All activities in the system—scheduling application programs, writing files to disk, sending data across a network, and so on—should function together as seamlessly and transparently as possible.

Some environments call for more rigorous resource management and scheduling than others. Realtime applications, for instance, depend on the OS to handle multiple events and to ensure that the system responds to those events within predictable time limits. The more responsive the OS, the more "time" a realtime application has to meet its deadlines.

The QNX Neutrino RTOS is ideal for *embedded realtime applications*. It can be scaled to very small sizes and provides multitasking, threads, priority-driven preemptive scheduling, and fast context-switching—all essential ingredients of an embedded realtime system. Moreover, the OS delivers these capabilities with a POSIX-standard API; there's no need to forgo standards in order to achieve a small system.

QNX Neutrino is also remarkably flexible. Developers can easily customize the OS to meet the needs of their applications. From a "bare-bones" configuration of the microkernel with a few small modules to a full-blown network-wide system equipped to serve hundreds of users, you're free to set up your system to use only those resources you require to tackle the job at hand.

QNX Neutrino achieves its unique degree of efficiency, modularity, and simplicity through two fundamental principles:

- microkernel architecture
- message-based interprocess communication

Microkernel architecture

Buzzwords often fall in and out of fashion. Vendors tend to enthusiastically apply the buzzwords of the day to their products, whether the terms actually fit or not.

The term “microkernel” has become fashionable. Although many new operating systems are said to be “microkernels” (or even “nanokernels”), the term may not mean very much without a clear definition.

Let's try to define the term. A microkernel OS is structured as a tiny kernel that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

The real goal in designing a microkernel OS is not simply to “make it small.” A microkernel OS embodies a fundamental change in the approach to delivering OS functionality. *Modularity is the key, size is but a side effect.* To call any kernel a “microkernel” simply because it happens to be small would miss the point entirely.

Since the IPC services provided by the microkernel are used to “glue” the OS itself together, the performance and flexibility of those services govern the performance of the resulting OS. With the exception of those IPC services, a microkernel is roughly comparable to a realtime executive, both in terms of the services provided and in their realtime performance.

The microkernel differs from an executive in how the IPC services are used to extend the functionality of the kernel with additional, service-providing processes. Since the OS is implemented as a team of cooperating processes managed by the microkernel, user-written processes can serve both as applications and as processes that extend the underlying OS functionality for industry-specific applications. The OS itself becomes “open” and easily extensible. Moreover, user-written extensions to the OS won't affect the fundamental reliability of the core OS.

A difficulty for many realtime executives implementing the POSIX 1003.1 standard is that their runtime environment is typically a single-process, multiple-threaded model, with unprotected memory between threads. Such an environment is only a subset of the multi-process model that POSIX assumes; it cannot support the *fork()* function. In contrast, QNX Neutrino fully utilizes an MMU to deliver the complete POSIX process model in a protected environment.

As the following diagrams show, a true microkernel offers *complete memory protection*, not only for user applications, but also for OS components (device drivers, filesystems, etc.):

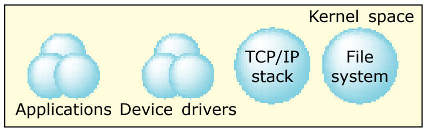


Figure 1: Conventional executives offer no memory protection.

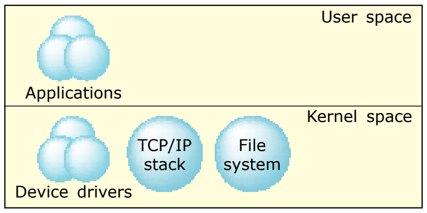


Figure 2: In a monolithic OS, system processes have no protection.

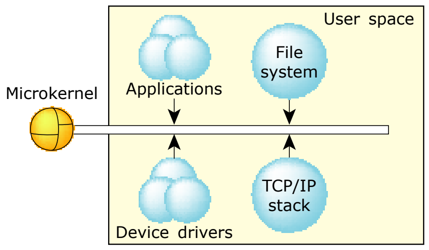


Figure 3: A microkernel provides complete memory protection.

The first version of the QNX OS was shipped in 1981. With each successive product revision, we have applied the experience from previous product generations to the latest incarnation, our most capable, scalable OS to date. We believe that this time-tested experience is what enables the QNX Neutrino RTOS to deliver the functionality it does using the limited resources it consumes.

The OS as a team of processes

The QNX Neutrino RTOS consists of a small microkernel managing a group of cooperating processes.

As the following illustration shows, the structure looks more like a team than a hierarchy, as several "players" of equal rank interact with each other through the coordinating kernel.

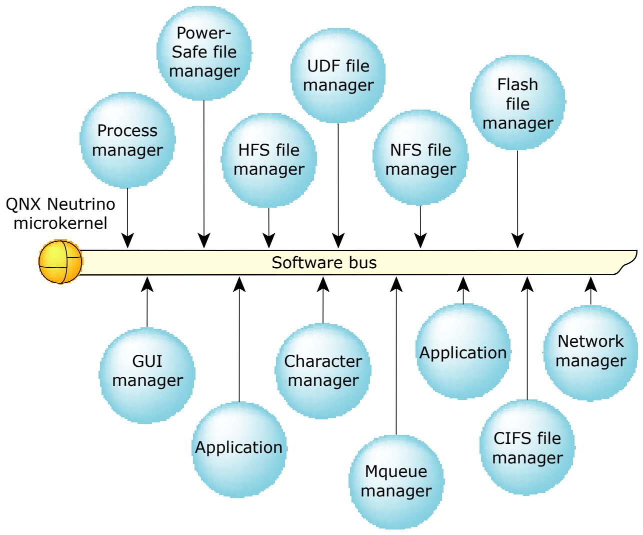


Figure 4: The QNX Neutrino RTOS architecture.

QNX Neutrino acts as a kind of "software bus" that lets you dynamically plug in/out OS modules whenever they're needed.

A true kernel

The *kernel* is the heart of any operating system. In some systems, the "kernel" comprises so many functions that for all intents and purposes it *is* the entire operating system!

But our microkernel is truly a kernel. First of all, like the kernel of a realtime executive, it's very small. Secondly, it's dedicated to only a few fundamental services:

- **thread services** via POSIX thread-creation primitives

- **signal services** via POSIX signal primitives
- **message-passing services**—the microkernel handles the routing of all messages between all threads throughout the entire system.
- **synchronization services** via POSIX thread-synchronization primitives.
- **scheduling services**—the microkernel schedules threads for execution using the various POSIX realtime scheduling policies.
- **timer services**—the microkernel provides the rich set of POSIX timer services.
- **process management services**—the microkernel and the process manager together form a unit (called `procnto`). The process manager portion is responsible for managing processes, memory, and the pathname space.

Unlike threads, the microkernel itself is never scheduled for execution. The processor executes code in the microkernel only as the result of an explicit kernel call, an exception, or in response to a hardware interrupt.

System processes

All OS services, except those provided by the mandatory microkernel/process manager module (`procnto`), are handled via *standard processes*.

A richly configured system could include the following:

- filesystem managers
- character device managers
- native network manager
- TCP/IP

System processes vs user-written processes

System processes are essentially indistinguishable from any user-written program—they use the same public API and kernel services available to any (suitably privileged) user process.

It is this architecture that gives the QNX Neutrino RTOS unparalleled extensibility. Since most OS services are provided by standard system processes, it's very simple to augment the OS itself: just write new programs to provide new OS services.

In fact, the boundary between the operating system and the application can become very blurred. The only real difference between system services and applications is that OS services manage resources for clients.

Suppose you've written a database server—how should such a process be classified?

Just as a filesystem accepts requests (via messages) to open files and read or write data, so too would a database server. While the requests to the database server may be more sophisticated, both servers are very much the same in that they provide an API (implemented by messages) that clients use to access a resource. Both are independent processes that can be written by an end-user and started and stopped on an as-needed basis.

A database server might be considered a system process at one installation, and an application at another. *It really doesn't matter!* The important point is that the OS allows such processes to be implemented cleanly, with no need for modifications to the standard components of the OS itself. For developers creating custom embedded systems, this provides the flexibility to extend the OS in directions that are uniquely useful to their applications, without needing access to OS source code.

Device drivers

Device drivers allow the OS and application programs to make use of the underlying hardware in a generic way (e.g., a disk drive, a network interface). While most OSs require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino can be started and stopped as standard processes. As a result, adding device drivers doesn't affect any other part of the OS—drivers can be developed and debugged like any other application.

Interprocess communication

When several threads run concurrently, as in typical realtime multitasking environments, the OS must provide mechanisms to allow them to communicate with each other.

Interprocess communication (IPC) is the key to designing an application as a set of cooperating processes in which each process handles one well-defined part of the whole.

The OS provides a simple but powerful set of IPC capabilities that greatly simplify the job of developing applications made up of cooperating processes. For more information, see the *Interprocess Communication (IPC)* chapter.

QNX Neutrino as a message-passing operating system

QNX Neutrino was the first commercial operating system of its kind to make use of message passing as the fundamental means of IPC. The OS owes much of its power, simplicity, and elegance to the complete integration of the message-passing method throughout the entire system.

In QNX Neutrino, a message is a parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message—the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various “changes of state” that affect when, and for how long, they may run. Knowing their states and priorities, the microkernel can schedule all processes as efficiently as possible to make the most of available CPU resources. This single, consistent method—message-passing—is thus constantly operative throughout the entire system.

Realtime and other mission-critical applications generally require a dependable form of IPC, because the processes that make up such applications are so strongly interrelated. The discipline imposed by QNX Neutrino's message-passing design helps bring order and greater reliability to applications.

Network distribution of kernels

In its simplest form, local area networking provides a mechanism for sharing files and peripheral devices among several interconnected computers. QNX Neutrino goes far beyond this simple concept and integrates the entire network into a single, homogeneous set of resources.

Any thread on any machine in the network can directly make use of any resource on any other machine. From the application's perspective, there's no difference between a local or remote resource—no special facilities need to be built into applications to allow them to make use of remote resources.

Users may access files anywhere on the network, take advantage of any peripheral device, and run applications on any machine on the network (provided they have the appropriate authority). Processes can communicate in the same manner anywhere throughout the entire network. Again, the OS's all-pervasive message-passing IPC accounts for such fluid, transparent networking.

Single-computer model

QNX Neutrino is designed from the ground up as a network-wide operating system.

In some ways, a native QNX Neutrino network feels more like a mainframe computer than a set of individual micros. Users are simply aware of a large set of resources available for use by any application. But unlike a mainframe, QNX Neutrino provides a highly responsive environment, since the appropriate amount of computing power can be made available at each node to meet the needs of each user.

In a mission-critical environment, for example, applications that control realtime I/O devices may require more performance than other, less critical, applications, such as a web browser. The network is responsive enough to support both types of applications *at the same time*—the OS lets you focus computing power on the devices in your hard realtime system where and when it's needed, without sacrificing concurrent connectivity to the desktop. Moreover, critical aspects of realtime computing, such as priority inheritance, function seamlessly across a QNX Neutrino network, regardless of the physical media employed (switch fabric, serial, etc.).

Flexible networking

QNX Neutrino networks can be put together using various hardware and industry-standard protocols. Since these are completely transparent to application programs and users, new network architectures can be introduced at any time without disturbing the OS.

Each node in the network is assigned a unique name that becomes its identifier. This name is the only visible means to determine whether the OS is running as a network or as a standalone operating system.

This degree of transparency is yet another example of the distinctive power of QNX Neutrino's message-passing architecture. In many systems, important functions such as networking, IPC, or even message passing are built on top of the OS, rather than integrated directly into its core. The result is often an awkward, inefficient "double standard" interface, whereby communication between processes is one thing, while penetrating the private interface of a mysterious monolithic kernel is another matter altogether.

In contrast to monolithic systems, QNX Neutrino is grounded on the principle that effective communication is the key to effective operation. Message passing thus forms the cornerstone of our microkernel architecture and enhances the efficiency of *all* transactions among all processes throughout the entire system, whether across a PC backplane or across a mile of twisted pair.

Chapter 2
The QNX Neutrino Microkernel

The microkernel implements the core POSIX features used in embedded realtime systems, along with the fundamental QNX Neutrino message-passing services.

The POSIX features that aren't implemented in the `procnto` microkernel (file and device I/O, for example) are provided by optional processes and shared libraries.

To determine the release version of the kernel on your system, use the `uname -a` command. For more information, see its entry in the *Utilities Reference*.

Successive microkernels from QNX Software Systems have seen a reduction in the code required to implement a given kernel call. The object definitions at the lowest layer in the kernel code have become more specific, allowing greater code reuse (such as folding various forms of POSIX signals, realtime signals, and QNX Neutrino pulses into common data structures and code to manipulate those structures).

At its lowest level, the microkernel contains a few fundamental objects and the highly tuned routines that manipulate them. The OS is built from this foundation.

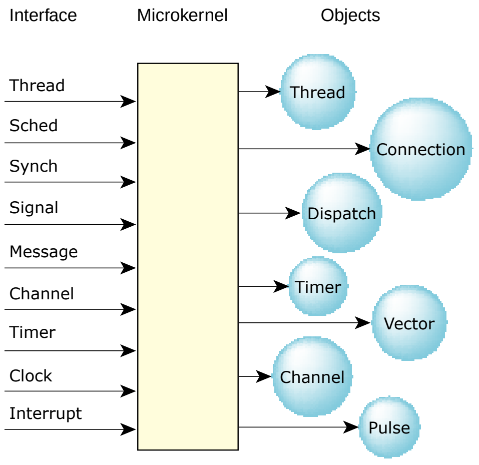


Figure 5: The microkernel.

Some developers have assumed that our microkernel is implemented entirely in assembly code for size or performance reasons. In fact, our implementation is coded primarily in C; size and performance goals are achieved through successively refined algorithms and data structures, rather than via assembly-level peep-hole optimizations.

The implementation of the QNX Neutrino RTOS

Historically, the “application pressure” on QNX Software Systems' operating systems has been from both ends of the computing spectrum—from memory-limited embedded systems all the way up to high-end SMP (symmetrical multiprocessing) machines with gigabytes of physical memory.

Accordingly, the design goals for QNX Neutrino accommodate both seemingly exclusive sets of functionality. Pursuing these goals is intended to extend the reach of systems well beyond what other OS implementations could address.

POSIX realtime and thread extensions

Since the QNX Neutrino RTOS implements the majority of the realtime and thread services directly in the microkernel, these services are available even without the presence of additional OS modules.

In addition, some of the profiles defined by POSIX suggest that these services be present without necessarily requiring a process model. In order to accommodate this, the OS provides direct support for threads, but relies on its process manager portion to extend this functionality to processes containing multiple threads.

Note that many realtime executives and kernels provide only a nonmemory-protected threaded model, with no process model and/or protected memory model at all. Without a process model, full POSIX compliance cannot be achieved.

System services

The microkernel has kernel calls to support the following:

- threads
- message passing
- signals
- clocks
- timers
- interrupt handlers
- semaphores
- mutual exclusion locks (mutexes)
- condition variables (condvars)
- barriers

The entire OS is built upon these calls. The OS is fully preemptible, even while passing messages between processes; it resumes the message pass where it left off before preemption.

The minimal complexity of the microkernel helps place an upper bound on the longest nonpreemptible code path through the kernel, while the small code size makes addressing complex multiprocessor issues a tractable problem. Services were chosen for inclusion in the microkernel on the basis of having a short execution path. Operations requiring significant work (e.g., process loading) were assigned to external processes/threads, where the effort to enter the context of that thread would be insignificant compared to the work done within the thread to service the request.

Rigorous application of this rule to dividing the functionality between the kernel and external processes destroys the myth that a microkernel OS must incur higher runtime overhead than a monolithic kernel OS. Given the work done between context switches (implicit in a message pass), and the very quick context-switch times that result from the simplified kernel, the time spent performing context switches becomes "lost in the noise" of the work done to service the requests communicated by the message passing between the processes that make up the OS.

The following diagram shows the preemption details for the non-SMP kernel (x86 implementation).

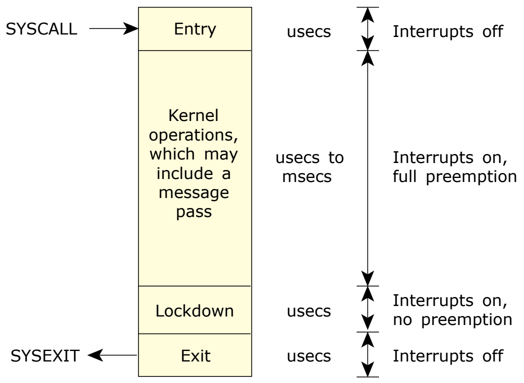

SYSCALL
Entry
usecs
Interrupts off
Kernel operations, which may include a message pass
usecs to msecs
Interrupts on, full preemption
Lockdown
usecs
Interrupts on, no preemption
SYSEXIT
Exit
usecs
Interrupts off

Figure 6: QNX Neutrino preemption details.

Interrupts are disabled, or preemption is held off, for only very brief intervals (typically in the order of hundreds of nanoseconds).

Threads and processes

When building an application (realtime, embedded, graphical, or otherwise), the developer may want several algorithms within the application to execute concurrently. This concurrency is achieved by using the POSIX thread model, which defines a process as containing one or more threads of execution.

A thread can be thought of as the minimum “unit of execution,” the unit of scheduling and execution in the microkernel. A process, on the other hand, can be thought of as a “container” for threads, defining the “address space” within which threads will execute. A process will always contain at least one thread.

Depending on the nature of the application, threads might execute independently with no need to communicate between the algorithms (unlikely), or they may need to be tightly coupled, with high-bandwidth communications and tight synchronization. To assist in this communication and synchronization, the QNX Neutrino RTOS provides a rich variety of IPC and synchronization services.

The following *pthread_** (POSIX Threads) library calls don't involve any microkernel thread calls:

- *pthread_attr_destroy()*
- *pthread_attr_getdetachstate()*
- *pthread_attr_getinheritsched()*
- *pthread_attr_getschedparam()*
- *pthread_attr_getschedpolicy()*
- *pthread_attr_getscope()*
- *pthread_attr_getstackaddr()*
- *pthread_attr_getstacksize()*
- *pthread_attr_init()*
- *pthread_attr_setdetachstate()*
- *pthread_attr_setinheritsched()*
- *pthread_attr_setschedparam()*
- *pthread_attr_setschedpolicy()*
- *pthread_attr_setscope()*
- *pthread_attr_setstackaddr()*
- *pthread_attr_setstacksize()*
- *pthread_cleanup_pop()*
- *pthread_cleanup_push()*
- *pthread_equal()*
- *pthread_getspecific()*
- *pthread_setspecific()*
- *pthread_key_create()*
- *pthread_key_delete()*
- *pthread_self()*

The following table lists the POSIX thread calls that have a corresponding microkernel thread call, allowing you to choose either interface:

POSIX call	Microkernel call	Description
pthread_create()	*ThreadCreate()*	Create a new thread of execution
pthread_exit()	*ThreadDestroy()*	Destroy a thread
pthread_detach()	*ThreadDetach()*	Detach a thread so it doesn't need to be joined
pthread_join()	*ThreadJoin()*	Join a thread waiting for its exit status
pthread_cancel()	*ThreadCancel()*	Cancel a thread at the next cancellation point
N/A	*ThreadCtl()*	Change a thread's QNX Neutrino-specific thread characteristics
pthread_mutex_init()	*SyncTypeCreate()*	Create a mutex
pthread_mutex_destroy()	*SyncDestroy()*	Destroy a mutex
pthread_mutex_lock()	*SyncMutexLock()*	Lock a mutex
pthread_mutex_trylock()	*SyncMutexLock()*	Conditionally lock a mutex
pthread_mutex_unlock()	*SyncMutexUnlock()*	Unlock a mutex
pthread_cond_init()	*SyncTypeCreate()*	Create a condition variable
pthread_cond_destroy()	*SyncDestroy()*	Destroy a condition variable
pthread_cond_wait()	*SyncCondvarWait()*	Wait on a condition variable
pthread_cond_signal()	*SyncCondvarSignal()*	Signal a condition variable
pthread_cond_broadcast()	*SyncCondvarSignal()*	Broadcast a condition variable
pthread_getschedparam()	*SchedGet()*	Get the scheduling parameters and policy of a thread
pthread_setschedparam(), *pthread_setschedprio()*	*SchedSet()*	Set the scheduling parameters and policy of a thread
pthread_sigmask()	*SignalProcmask()*	Examine or set a thread's signal mask
pthread_kill()	*SignalKill()*	Send a signal to a specific thread

The OS can be configured to provide a mix of threads and processes (as defined by POSIX). Each process is MMU-protected from each other, and each process may contain one or more threads that share the process's address space.

The environment you choose affects not only the concurrency capabilities of the application, but also the IPC and synchronization services the application might make use of.

Even though the common term “IPC” refers to communicating processes, we use it here to describe the communication between *threads*, whether they're within the same process or separate processes.

For information about processes and threads from the programming point of view, see the Processes and Threads chapter of *Getting Started with QNX Neutrino*, and the Programming Overview and Processes chapters of the QNX Neutrino *Programmer's Guide*.

Thread attributes

Although threads within a process share everything within the process's address space, each thread still has some “private” data. In some cases, this private data is protected within the kernel (e.g., the *tid* or thread ID), while other private data resides unprotected in the process's address space (e.g., each thread has a stack for its own use). Some of the more noteworthy thread-private resources are:

tid

Each thread is identified by an integer thread ID, starting at 1. The *tid* is unique within the thread's process.

Priority

Each thread has a priority that helps determine when it runs. A thread inherits its initial priority from its parent, but the priority can change, depending on the scheduling policy, explicit changes that the thread makes, or messages sent to the thread.

In the QNX Neutrino RTOS, processes don't have priorities; their threads do.

For more information, see “*Thread scheduling*,” later in this chapter.

Name

Starting with the QNX Neutrino Core OS 6.3.2, you can assign a name to a thread; see the entries for *pthread_getname_np()* and *pthread_setname_np()* in the QNX Neutrino *C Library Reference*. Utilities such as `dumper` and `pidin` support thread names. Thread names are a QNX Neutrino extension.

Register set

Each thread has its own instruction pointer (IP), stack pointer (SP), and other processor-specific register context.

Stack

Each thread executes on its own stack, stored within the address space of its process.

Signal mask

Each thread has its own signal mask.

Thread local storage

A thread has a system-defined data area called “thread local storage” (TLS). The TLS is used to store “per-thread” information (such as *tid*, *pid*, stack base, *errno*, and thread-specific key/data bindings). The TLS doesn't need to be accessed directly by a user application. A thread can have user-defined data associated with a thread-specific data key.

Cancellation handlers

Callback functions that are executed when the thread terminates.

Thread-specific data, implemented in the *pthread* library and stored in the TLS, provides a mechanism for associating a process global integer key with a unique per-thread data value. To use thread-specific data, you first create a new key and then bind a unique data value to the key (per thread). The data value may, for example, be an integer or a pointer to a dynamically allocated data structure. Subsequently, the key can return the bound data value per thread.

A typical application of thread-specific data is for a thread-safe function that needs to maintain a context for each calling thread.

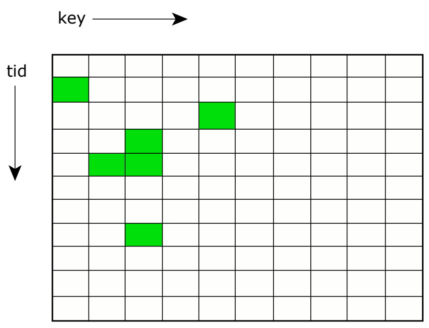


Figure 7: Sparse matrix (`tid,key`) to value mapping.

You use the following functions to create and manipulate this data:

Function	Description
pthread_key_create()	Create a data key with destructor function
pthread_key_delete()	Destroy a data key
pthread_setspecific()	Bind a data value to a data key
pthread_getspecific()	Return the data value bound to a data key

Thread life cycle

The number of threads within a process can vary widely, with threads being created and destroyed dynamically.

Thread creation (*pthread_create()*) involves allocating and initializing the necessary resources within the process's address space (e.g., thread stack) and starting the execution of the thread at some function in the address space.

Thread termination (*pthread_exit()*, *pthread_cancel()*) involves stopping the thread and reclaiming the thread's resources. As a thread executes, its state can generally be described as either "ready" or "blocked." More specifically, it can be one of the following:

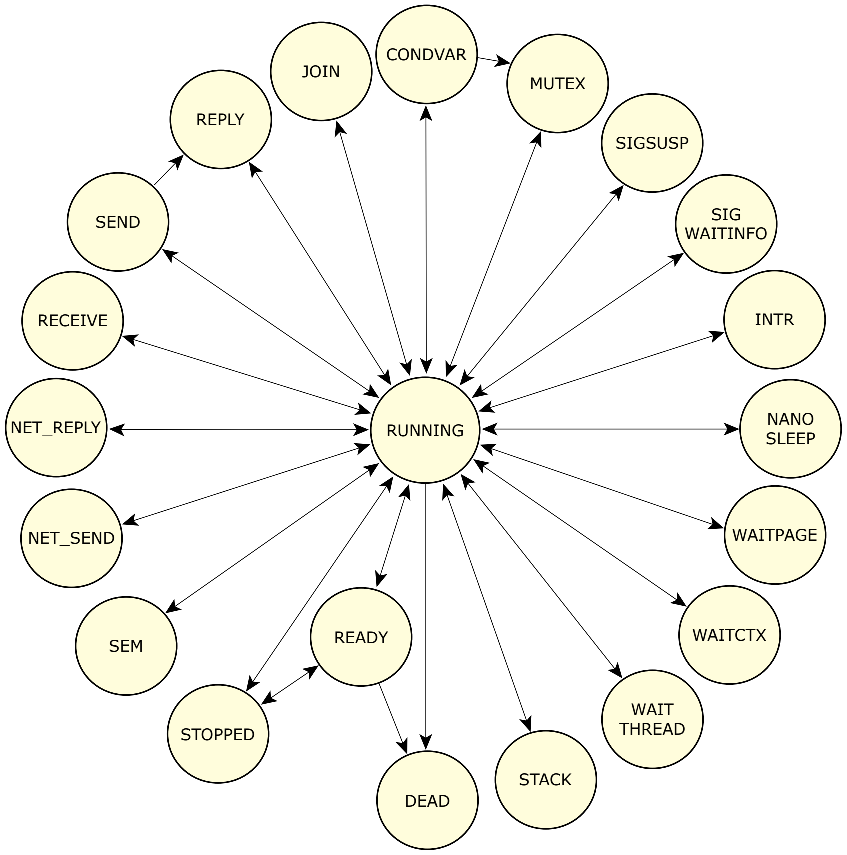


Figure 8: Possible thread states. Note that, in addition to the transitions shown above, a thread can move from any state (except DEAD) to READY.

STATE_CONDVAR

The thread is blocked on a condition variable (e.g., it called *pthread_cond_wait()*).

STATE_DEAD

The thread has terminated and is waiting for a join by another thread.

STATE_INTR

The thread is blocked waiting for an interrupt (i.e., it called *InterruptWait()*).

STATE_JOIN

The thread is blocked waiting to join another thread (e.g., it called *pthread_join()*).

STATE_MUTEX

The thread is blocked on a mutual exclusion lock (e.g., it called *pthread_mutex_lock()*).

STATE_NANOSLEEP

The thread is sleeping for a short time interval (e.g., it called *nanosleep()*).

STATE_NET_REPLY

The thread is waiting for a reply to be delivered across the network (i.e., it called *MsgReply*()*).

STATE_NET_SEND

The thread is waiting for a pulse or signal to be delivered across the network (i.e., it called *MsgSendPulse()*, *MsgDeliverEvent()*, or *SignalKill()*).

STATE_READY

The thread is waiting to be executed while the processor executes another thread of equal or higher priority.

STATE_RECEIVE

The thread is blocked on a message receive (e.g., it called *MsgReceive()*).

STATE_REPLY

The thread is blocked on a message reply (i.e., it called *MsgSend()*, and the server received the message).

STATE_RUNNING

The thread is being executed by a processor. The kernel uses an array (with one entry per processor in the system) to keep track of the running threads.

STATE_SEM

The thread is waiting for a semaphore to be posted (i.e., it called *SyncSemWait()*).

STATE_SEND

The thread is blocked on a message send (e.g., it called *MsgSend()*, but the server hasn't yet received the message).

STATE_SIGSUSPEND

The thread is blocked waiting for a signal (i.e., it called *sigsuspend()*).

STATE_SIGWAITINFO

The thread is blocked waiting for a signal (i.e., it called *sigwaitinfo()*).

STATE_STACK

The thread is waiting for the virtual address space to be allocated for the thread's stack (parent will have called *ThreadCreate()*).

STATE_STOPPED

The thread is blocked waiting for a SIGCONT signal.

STATE_WAITCTX

The thread is waiting for a noninteger (e.g., floating point) context to become available for use.

STATE_WAITPAGE

The thread is waiting for physical memory to be allocated for a virtual address.

STATE_WAITTHREAD

The thread is waiting for a child thread to finish creating itself (i.e., it called *ThreadCreate()*).

In discussion and in the documentation, we usually omit the “STATE_” prefix.

Thread scheduling

When and how are scheduling decisions made?

The microkernel makes scheduling decisions whenever it's entered as the result of a kernel call, exception, or hardware interrupt. A scheduling decision is made whenever the execution state of any thread changes—it doesn't matter which processes the threads might reside within. Threads are scheduled globally across all processes.

Normally the running thread continues to run, but the thread scheduler will perform a context switch from one thread to another whenever the running thread:

- blocks
- is preempted
- yields

When does a thread block?

The running thread blocks when it must wait for some event to occur (response to an IPC request, wait on a mutex, etc.). The blocked thread is removed from the running array and the highest-priority ready thread is then run. When the blocked thread is subsequently unblocked, it's usually placed on the end of the ready queue for that priority level.

When is a thread preempted?

The running thread is preempted when a higher-priority thread is placed on the ready queue (it becomes READY, as the result of its block condition being resolved). The preempted thread is put at the beginning of the ready queue for that priority and the higher-priority thread runs.

When does a thread yield?

The running thread voluntarily yields the processor (e.g., via (*sched_yield()*) and is placed on the end of the ready queue for that priority. The highest-priority thread then runs (which may still be the thread that just yielded).

Scheduling priority

Every thread is assigned a priority. The thread scheduler selects the next thread to run by looking at the priority assigned to every thread that's READY (i.e., capable of using the CPU).

- On a single-core system, the READY thread with the highest priority is selected to run.
- On a multicore (SMP) system, the scheduler runs the highest-priority READY thread on one of the available cores. Additional cores run other threads in the system, though not necessarily the next highest-priority thread or threads; the scheduler is allowed some flexibility so that it can try to optimize cache usage and minimize thread migration. The next time a processor that's running a lower-priority thread makes a scheduling decision, it will choose the higher-priority one. Scheduling is also affected by *processor affinity*, which threads can use to specify which processors they can run on. For more information, see the *Multicore Processing* chapter.

The following diagram shows the ready queue for a single-core system with five threads (B–F) that are READY. Thread A is currently running. All other threads (G–Z) are BLOCKED. Thread A, B, and C are at the highest priority, so they'll share the processor based on the running thread's scheduling policy.

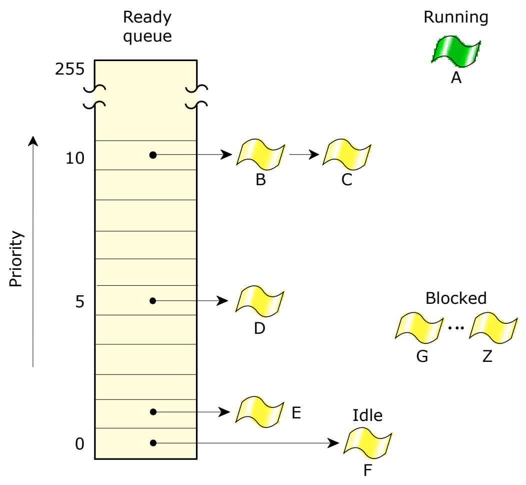


Figure 9: The ready queue in a single-core system.

The OS supports a total of 256 scheduling priority levels. An unprivileged thread can set its priority to a level from 1 to 63 (the highest unprivileged priority), *independent of the scheduling policy*. Only threads with the PROCMGR_AID_PRIORITY ability enabled (see *procmgr_ability()* in the *C Library Reference*) are allowed to set priorities above 63. The special *idle* thread (in the process manager) has priority 0 and is always ready to run. A thread inherits the priority of its parent thread by default.

You can change the allowed priority range for unprivileged processes with the −P option for `procnto`:

```
procnto-smp-instr -P priority
```

In QNX Neutrino 6.6 or later, you can append an `s` or `S` to this option if you want out-of-range priority requests by default to use the maximum allowed priority (reach a “maximum saturation point”) instead of resulting in an error. When you're setting a priority, you can wrap it in one these (non-POSIX) macros to specify how to handle out-of-range priority requests:

- *SCHED_PRIO_LIMIT_ERROR(priority)*—indicate an error
- *SCHED_PRIO_LIMIT_SATURATE(priority)*—use the maximum allowed priority

Here's a summary of the ranges:

Priority level	Owner
0	Idle thread
1 through *priority* – 1	Unprivileged or privileged
priority through 255	Privileged

Note that in order to prevent *priority inversion*, the kernel may temporarily boost a thread's priority. For more information, see “*Priority inheritance and mutexes*” later in this chapter, and “*Priority inheritance and messages*” in the Interprocess Communication (IPC) chapter. The initial priority of the kernel's threads is 255, but the first thing they all do is block in a *MsgReceive()*, so after that they operate at the priority of threads that send messages to them.

The threads on the ready queue are ordered by priority. The ready queue is actually implemented as 256 separate queues, one for each priority. The first thread in the highest-priority queue is selected to run.

Most of the time, threads are queued in FIFO order in the queue of their priority, but there are some exceptions:

- A server thread that's coming out of a RECEIVE-blocked state with a message from a client is inserted at the head of the queue for that priority—that is, the order is LIFO, not FIFO.
- If a thread sends a message with an “nc” (non-cancellation point) variant of *MsgSend*()*, then when the server replies, the thread is placed at the front of the ready queue, rather than at the end. If the scheduling policy is round-robin, the thread's timeslice isn't replenished; for example, if the thread had already used half its timeslice before sending, then it still has only half a timeslice left before being eligible for preemption.

Scheduling policies

To meet the needs of various applications, the QNX Neutrino RTOS provides these scheduling algorithms:

- FIFO scheduling
- round-robin scheduling
- sporadic scheduling

Each thread in the system may run using any method. The methods are effective on a per-thread basis, not on a global basis for all threads and processes on a node.

Remember that the FIFO and round-robin scheduling policies apply only when two or more threads that share the *same priority* are READY (i.e., the threads are directly competing with each other). The sporadic method, however, employs a “budget” for a thread's execution. In all cases, if a higher-priority thread becomes READY, it immediately preempts all lower-priority threads.

In the following diagram, three threads of equal priority are READY. If Thread A blocks, Thread B will run.

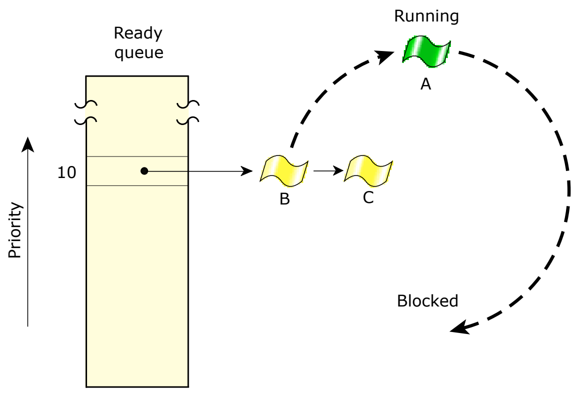


Figure 10: Thread A blocks; Thread B runs.

Although a thread inherits its scheduling policy from its parent process, the thread can request to change the algorithm applied by the kernel.

FIFO scheduling

In FIFO scheduling, a thread selected to run continues executing until it:

- voluntarily relinquishes control (e.g., it blocks)
- is preempted by a higher-priority thread

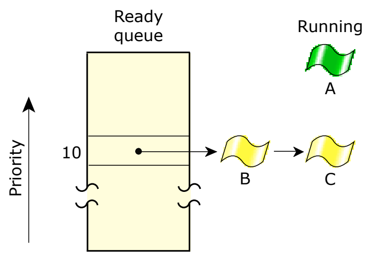


Figure 11: FIFO scheduling.

Round-robin scheduling

In round-robin scheduling, a thread selected to run continues executing until it:

- voluntarily relinquishes control
- is preempted by a higher-priority thread
- consumes its *timeslice*

As the following diagram shows, Thread A ran until it consumed its timeslice; the next READY thread (Thread B) now runs:

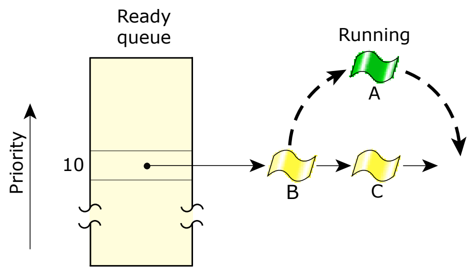


Figure 12: Round-robin scheduling.

A timeslice is the unit of time assigned to every process. Once it consumes its timeslice, a thread is preempted and the next READY thread at the same priority level is given control. A timeslice is 4 × the clock period. (For more information, see the entry for *ClockPeriod()* in the QNX Neutrino *C Library Reference*.)

Apart from time slicing, round-robin scheduling is identical to FIFO scheduling.

Sporadic scheduling

The sporadic scheduling policy is generally used to provide a capped limit on the execution time of a thread *within a given period of time*.

This behavior is essential when Rate Monotonic Analysis (RMA) is being performed on a system that services both periodic and aperiodic events. Essentially, this algorithm allows a thread to service aperiodic events without jeopardizing the hard deadlines of other threads or processes in the system.

As in FIFO scheduling, a thread using sporadic scheduling continues executing until it blocks or is preempted by a higher-priority thread. And as in adaptive scheduling, a thread using sporadic scheduling will drop in priority, but with sporadic scheduling you have much more precise control over the thread's behavior.

Under sporadic scheduling, a thread's priority can oscillate dynamically between a *foreground* or normal priority and a *background* or low priority. Using the following parameters, you can control the conditions of this sporadic shift:

Initial budget (C)

The amount of time a thread is allowed to execute at its normal priority (N) before being dropped to its low priority (L).

Low priority (L)

The priority level to which the thread will drop. The thread executes at this lower priority (L) while in the background, and runs at normal priority (N) while in the foreground.

Replenishment period (T)

The period of time during which a thread is allowed to consume its execution budget. To schedule replenishment operations, the POSIX implementation also uses this value as the offset from the time the thread becomes READY.

Max number of pending replenishments

This value limits the number of replenishment operations that can take place, thereby bounding the amount of system overhead consumed by the sporadic scheduling policy.

In a poorly configured system, a thread's execution budget may become eroded because of too much blocking—i.e., it won't receive enough replenishments.

As the following diagram shows, the sporadic scheduling policy establishes a thread's initial execution budget (C), which is consumed by the thread as it runs and is replenished periodically (for the amount T). When a thread blocks, the amount of the execution budget that's been consumed (R) is arranged to be replenished at some later time (e.g., at 40 msec) after the thread first became ready to run.

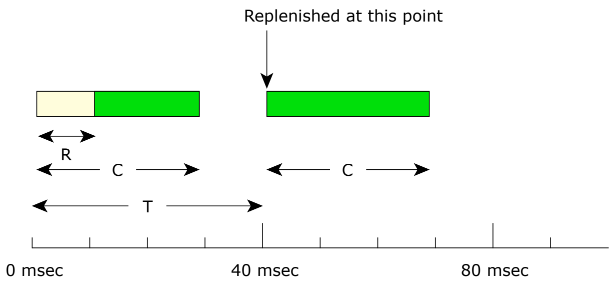


Figure 13: A thread's budget is replenished periodically.

At its normal priority N, a thread will execute for the amount of time defined by its initial execution budget C. As soon as this time is exhausted, the priority of the thread will drop to its low priority L until the replenishment operation occurs.

Assume, for example, a system where the thread never blocks or is never preempted:

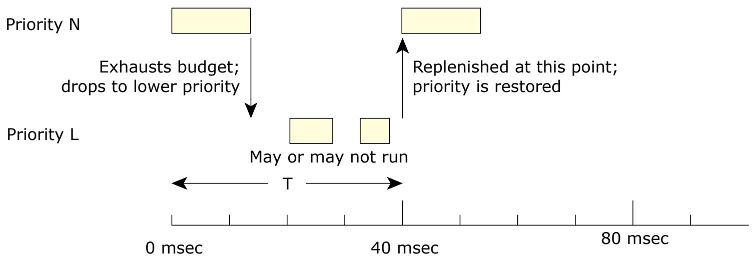


Figure 14: A thread drops in priority until its budget is replenished.

Here the thread will drop to its low-priority (background) level, where it may or may not get a chance to run depending on the priority of other threads in the system.

Once the replenishment occurs, the thread's priority is raised to its original level. This guarantees that within a properly configured system, the thread will be given the opportunity *every period T* to run for a maximum execution time C. This ensures that a thread running at priority N will consume only C/T of the system's resources.

The QNX Neutrino implementation of the sporadic scheduler differs from the POSIX sporadic scheduler in that, for computational reasons, the algorithm allows at most one pending replenishment per sporadic server thread. At each replenishment, the available budget of the sporadic server thread gets boosted up to its initial budget.

Manipulating priority and scheduling policies

A thread's priority can vary during its execution, either from direct manipulation by the thread itself or from the kernel adjusting the thread's priority as it receives a message from a higher-priority thread.

In addition to priority, you can also select the scheduling algorithm that the kernel will use for the thread. Although our libraries provide a number of different ways to get and set the scheduling parameters, your best choices are *pthread_getschedparam()*, *pthread_setschedparam()*, and *pthread_setschedprio()*. For information about the other choices, see “Scheduling policies” in the Programming Overview chapter of the QNX Neutrino *Programmer's Guide*.

IPC issues

Since all the threads in a process have unhindered access to the shared data space, wouldn't this execution model “trivially” solve all of our IPC problems? Can't we just communicate the data through shared memory and dispense with any other execution models and IPC mechanisms?

If only it were that simple!

One issue is that the access of individual threads to common data must be *synchronized*. Having one thread read inconsistent data because another thread is part way through modifying it is a recipe for disaster. For example, if one thread is updating a linked list, no other threads can be allowed to traverse or modify the list until the first thread has finished. A code passage that must execute “serially” (i.e., by only one thread at a time) in this manner is termed a “critical section.” The program would fail (intermittently, depending on how frequently a “collision” occurred) with irreparably damaged links unless some synchronization mechanism ensured serial access.

Mutexes, semaphores, and condvars are examples of synchronization tools that can be used to address this problem. These tools are described later in this section.

Although synchronization services can be used to allow threads to cooperate, shared memory per se can't address a number of IPC issues. For example, although threads can communicate through the common data space, this works only if all the threads communicating are within a single process. What if our application needs to communicate a query to a database server? We need to pass the details of our query to the database server, but the thread we need to communicate with lies *within* a database server process and the address space of that server isn't addressable to us.

The OS takes care of the network-distributed IPC issue because the one interface—message passing—operates in both the local and network-remote cases, and can be used to access all OS services. Since messages can be exactly sized, and since most messages tend to be quite tiny (e.g., the error status on a write request, or a tiny read request), the data moved around the network can be far less with message passing than with network-distributed shared memory, which would tend to copy 4K pages around.

Thread complexity issues

Although threads are very appropriate for some system designs, it's important to respect the Pandora's box of complexities their use unleashes.

In some ways, it's ironic that while MMU-protected multitasking has become common, computing fashion has made popular the use of multiple threads in an unprotected address space. This not only makes debugging difficult, but also hampers the generation of reliable code.

Threads were initially introduced to UNIX systems as a “light-weight” concurrency mechanism to address the problem of slow context switches between “heavy weight” processes. Although this is a worthwhile goal, an obvious question arises: Why are process-to-process context switches slow in the first place?

Architecturally, the OS addresses the context-switch performance issue first. In fact, threads and processes provide nearly identical context-switch performance numbers. The QNX Neutrino RTOS's process-switch times are faster than UNIX thread-switch times. As a result, QNX Neutrino threads don't need to be used to solve the IPC performance problem; instead, they're a tool for achieving greater concurrency within application and server processes.

Without resorting to threads, fast process-to-process context switching makes it reasonable to structure an application as a team of cooperating processes sharing an explicitly allocated shared-memory region. An application thus exposes itself to bugs in the cooperating processes only so far as the effects of those bugs on the contents of the shared-memory region. The private memory of the process is still protected from the other processes. In the purely threaded model, the private data of all threads (including their stacks) is openly accessible, vulnerable to stray pointer errors in any thread in the process.

Nevertheless, threads can also provide concurrency advantages that a pure process model cannot address. For example, a filesystem server process that executes requests on behalf of many clients (where each request takes significant time to complete), definitely benefits from having multiple threads of execution. If one client process requests a block from disk, while another client requests a block already in cache, the filesystem process can utilize a pool of threads to concurrently service client requests, rather than remain "busy" until the disk block is read for the first request.

As requests arrive, each thread is able to respond directly from the buffer cache or to block and wait for disk I/O without increasing the response latency seen by other client processes. The filesystem server can "precreate" a team of threads, ready to respond in turn to client requests as they arrive. Although this complicates the architecture of the filesystem manager, the gains in concurrency are significant.

Synchronization services

The QNX Neutrino RTOS provides the POSIX-standard thread-level synchronization primitives, some of which are useful even between threads in different processes.

The synchronization services include at least the following:

Synchronization service	Supported between processes	Supported across a QNX Neutrino LAN
Mutexes	Yes[a]	No
Condvars	Yes	No
Barriers	Yes[a]	No
Sleepon locks	No	No
Reader/writer locks	Yes[a]	No
Semaphores	Yes	Yes (named only)
FIFO scheduling	Yes	No
Send/Receive/Reply	Yes	Yes
Atomic operations	Yes	No

[a] Sharing this type of object between processes can be a security problem; see "*Safely sharing mutexes, barriers, and reader/writer locks between processes*," later in this chapter.

The above synchronization primitives are implemented directly by the kernel, except for:

- barriers, sleepon locks, and reader/writer locks (which are built from mutexes and condvars)
- atomic operations (which are either implemented directly by the processor or emulated in the kernel)

You should allocate mutexes, condvars, barriers, reader/writer locks, and semaphores, as well as objects you plan to use atomic operations on, only in normal memory mappings. On certain processors, atomic operations and calls such as *pthread_mutex_lock()* will cause a fault if the object is allocated in uncached memory.

Mutexes: mutual exclusion locks

Mutual exclusion locks, or *mutexes*, are the simplest of the synchronization services. A mutex is used to ensure exclusive access to data shared between threads.

A mutex is typically acquired (*pthread_mutex_lock()* or *pthread_mutex_timedlock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a critical section).

Only one thread may have the mutex locked at any given time. Threads attempting to lock an already locked mutex will block until the thread that owns the mutex unlocks it. When the thread unlocks the mutex, the highest-priority thread waiting to lock the mutex will unblock and become the new owner of the mutex. In this way, threads will sequence through a critical region in priority-order.

In most situations, acquisition of a mutex doesn't require entry to the kernel for a free mutex. What allows this is the use of the compare-and-swap opcode on x86 processors and the load/store conditional opcodes on most RISC processors.

Entry to the kernel is necessary at acquisition time if the mutex is already held, so that the thread can go on a blocked list; kernel entry is done on exit if other threads are waiting to be unblocked on that mutex. This allows normal acquisition and release of an uncontested critical section or resource to be very quick, incurring work by the OS only to resolve contention.

A nonblocking lock function (*pthread_mutex_trylock()*) can be used to test whether the mutex is currently locked or not. For best performance, the execution time of the critical section should be small and of bounded duration. A condvar should be used if the thread may block within the critical section.

Priority inheritance and mutexes

By default, if a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the higher-priority blocked thread waiting for the mutex (but see below). The current owner's effective priority is again adjusted when it unlocks the mutex; its new priority is the maximum of its own priority and the priorities of those threads it still blocks, either directly or indirectly.

This scheme not only ensures that the higher-priority thread will be blocked waiting for the mutex for the shortest possible time, but also solves the classic priority-inversion problem.

The *pthread_mutexattr_init()* function sets the protocol to PTHREAD_PRIO_INHERIT to allow this behavior; you can call *pthread_mutexattr_setprotocol()* to override this setting. The *pthread_mutex_trylock()* function doesn't change the thread priorities because it doesn't block.

What happens if the thread that's waiting for the mutex is running at a privileged priority, and the mutex owner's process doesn't have the PROCMGR_AID_PRIORITY ability enabled? In this case, the thread that owns the mutex is boosted to the highest unprivileged priority. For more information, see "*Scheduling priority*" earlier in this chapter and *procmgr_ability()* in the *C Library Reference*.

Other attributes

You can also modify other mutex attributes before initializing a mutex:

- Use *pthread_mutexattr_settype()* to allow a mutex to be recursively locked by the same thread. This can be useful to allow a thread to call a routine that might attempt to lock a mutex that the thread already happens to have locked.
- (QNX Neutrino 7.0 or later) Use *pthread_mutexattr_setrobust()* to set the mutex's robustness, which helps you recover the mutex if its owner terminates while holding it.

  The non-POSIX *SyncMutexEvent()* kernel call provides a different (and mutually exclusive) mechanism for recovering a mutex.
- Use *pthread_mutexattr_setprioceiling()* to set the priority ceiling.

Note that robust mutexes, mutexes with priority ceilings, and those using *SyncMutexEvent()* use more system resources than other mutexes.

Condvars: condition variables

A condition variable, or *condvar*, is used to block a thread within a critical section until some condition is satisfied. The condition can be arbitrarily complex and is independent of the condvar. However, the condvar must always be used with a mutex lock in order to implement a monitor.

A condvar supports three operations:

- wait (*pthread_cond_wait()*)
- signal (*pthread_cond_signal()*)
- broadcast (*pthread_cond_broadcast()*)

Note that there's no connection between a condvar signal and a POSIX signal.

Here's a typical example of how a condvar can be used:

```
pthread_mutex_lock( &m );
. . .
while (!arbitrary_condition) {
    pthread_cond_wait( &cv, &m );
    }
. . .
pthread_mutex_unlock( &m );
```

In this code sample, the mutex is acquired before the condition is tested. This ensures that only this thread has access to the arbitrary condition being examined. While the condition is true, the code sample will block on the wait call until some other thread performs a signal or broadcast on the condvar.

The `while` loop is required for two reasons. First of all, POSIX cannot guarantee that false wakeups will not occur (e.g., multiprocessor systems). Second, when another thread has made a modification to the condition, we need to retest to ensure that the modification matches our criteria. The associated mutex is unlocked atomically by *pthread_cond_wait()* when the waiting thread is blocked to allow another thread to enter the critical section.

A thread that performs a signal will unblock the highest-priority thread queued on the condvar, while a broadcast will unblock all threads queued on the condvar. The associated mutex is locked atomically by the highest-priority unblocked thread; the thread must then unlock the mutex after proceeding through the critical section.

A version of the condvar wait operation allows a timeout to be specified (*pthread_cond_timedwait()*). The waiting thread can then be unblocked when the timeout expires.

An application shouldn't use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a *pthread_cond_wait()* or *pthread_cond_timedwait()*

may not actually release the mutex (if it's been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

Barriers

A barrier is a synchronization mechanism that lets you "corral" several cooperating threads (e.g., in a matrix computation), forcing them to wait at a specific point until all have finished before any one thread can continue.

Unlike the *pthread_join()* function, where you'd wait for the threads to terminate, in the case of a barrier you're waiting for the threads to *rendezvous* at a certain point. When the specified number of threads arrive at the barrier, we unblock *all of them* so they can continue to run.

You first create a barrier with *pthread_barrier_init()*:

```
#include <pthread.h>

int
pthread_barrier_init (pthread_barrier_t *barrier,
                      const pthread_barrierattr_t *attr,
                      unsigned int count);
```

This creates a barrier object at the passed address (a pointer to the barrier object is in *barrier*), with the attributes as specified by *attr*. The *count* member holds the number of threads that must call *pthread_barrier_wait()*.

Once the barrier is created, each thread will call *pthread_barrier_wait()* to indicate that it has completed:

```
#include <pthread.h>

int pthread_barrier_wait (pthread_barrier_t *barrier);
```

When a thread calls *pthread_barrier_wait()*, it blocks until the number of threads specified initially in the *pthread_barrier_init()* function have called *pthread_barrier_wait()* (and blocked also). When the correct number of threads have called *pthread_barrier_wait()*, all those threads will unblock *at the same time*.

Here's an example:

```
/*
 *  barrier1.c
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <sys/neutrino.h>

pthread_barrier_t   barrier; // barrier synchronization object
```

```
void *
thread1 (void *not_used)
{
    time_t  now;

    time (&now);
    printf ("thread1 starting at %s", ctime (&now));

    // do the computation
    // let's just do a sleep here...
    sleep (20);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread1() done at %s", ctime (&now));
}

void *
thread2 (void *not_used)
{
    time_t  now;

    time (&now);
    printf ("thread2 starting at %s", ctime (&now));

    // do the computation
    // let's just do a sleep here...
    sleep (40);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread2() done at %s", ctime (&now));
}

int main () // ignore arguments
{
    time_t  now;

    // create a barrier object with a count of 3
    pthread_barrier_init (&barrier, NULL, 3);

    // start up two threads, thread1 and thread2
    pthread_create (NULL, NULL, thread1, NULL);
    pthread_create (NULL, NULL, thread2, NULL);

    // at this point, thread1 and thread2 are running

    // now wait for completion
    time (&now);
    printf ("main() waiting for barrier at %s", ctime (&now));
    pthread_barrier_wait (&barrier);

    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in main() done at %s", ctime (&now));
    pthread_exit( NULL );
    return (EXIT_SUCCESS);
}
```

The main thread created the barrier object and initialized it with a count of the total number of threads that must be synchronized to the barrier before the threads may carry on. In the example above, we used a count of 3: one for the *main()* thread, one for *thread1()*, and one for *thread2()*.

Then we start *thread1()* and *thread2()*. To simplify this example, we have the threads sleep to cause a delay, as if computations were occurring. To synchronize, the main thread simply blocks itself on the barrier, knowing that the barrier will unblock only after the two worker threads have joined it as well.

In this release, the following barrier functions are included:

Function	Description
pthread_barrierattr_getpshared()	Get the value of a barrier's process-shared attribute
pthread_barrierattr_destroy()	Destroy a barrier's attributes object
pthread_barrierattr_init()	Initialize a barrier's attributes object
pthread_barrierattr_setpshared()	Set the value of a barrier's process-shared attribute
pthread_barrier_destroy()	Destroy a barrier
pthread_barrier_init()	Initialize a barrier
pthread_barrier_wait()	Synchronize participating threads at the barrier

Sleepon locks

Sleepon locks are very similar to condvars, with a few subtle differences.

Like condvars, sleepon locks (*pthread_sleepon_lock()*) can be used to block until a condition becomes true (like a memory location changing value). But unlike condvars, which must be allocated for each condition to be checked, sleepon locks multiplex their functionality over a single mutex and dynamically allocated condvar, regardless of the number of conditions being checked. The maximum number of condvars ends up being equal to the maximum number of blocked threads. These locks are patterned after the sleepon locks commonly used within the UNIX kernel.

Reader/writer locks

More formally known as “Multiple readers, single writer locks,” these locks are used when the access pattern for a data structure consists of many threads reading the data, and (at most) one thread writing the data. These locks are more expensive than mutexes, but can be useful for this data access pattern.

This lock works by allowing all the threads that request a read-access lock (*pthread_rwlock_rdlock()*) to succeed in their request. But when a thread wishing to write asks for the lock (*pthread_rwlock_wrlock()*), the request is denied until all the current reading threads release their reading locks (*pthread_rwlock_unlock()*).

Multiple writing threads can queue (in priority order) waiting for their chance to write the protected data structure, and all the blocked writer-threads will get to run before reading threads are allowed access again. The priorities of the reading threads are not considered.

There are also calls (*pthread_rwlock_tryrdlock()* and *pthread_rwlock_trywrlock()*) to allow a thread to test the attempt to achieve the requested lock, without blocking. These calls return with a successful lock or a status indicating that the lock couldn't be granted immediately.

Reader/writer locks aren't implemented directly within the kernel, but are instead built from the mutex and condvar services provided by the kernel.

Safely sharing mutexes, barriers, and reader/writer locks between processes

You can share most synchronization objects between processes, but security can be a concern.

Most of this discussion involves mutexes, but barriers and reader/writer locks are built from mutexes, so it applies to them too.

In order for processes to share a synchronization object, they must also share the memory in which it resides.

The problem with shared mutexes is that a thread in any process can claim that another thread (in any process) owns the mutex; when priority inheritance is applied, the latter's priority is boosted to that of the former. If an application needs to share a mutex between separate processes, then it must decide whether to expose itself to potential interference from unrelated processes, disable priority inheritance on the shared mutex, or force all operations on the shared mutex to enter the kernel—which they don't normally do—resulting in a performance penalty.

In QNX Neutrino 7.0.1 or later, you can configure the kernel to reject attempts to lock shared mutexes that would cause priority inheritance unless all mutex locking operations enter the kernel. This has a significant impact on the performance of shared mutexes that use the priority-inheritence protocol, but guarantees that noncooperating threads can't interfere with each other. In order to use a shared mutex in a safe manner, you must do the following:

- Specify the `-s` option for `procnto` to force all mutex operations on shared mutexes that support priority inheritance to enter the kernel.
- Either disable priority inheritance for the mutex by using *pthread_mutexattr_setprotocol()* to set the PTHREAD_PRIO_NONE flag, or explicitly identify the mutex as shared by using *pthread_mutexattr_setpshared()* to set the process-shared attribute to PTHREAD_PROCESS_SHARED.

If you specify the `-s` option, the kernel rejects any attempts to lock a PTHREAD_PRIO_INHERIT mutex that doesn't have PTHREAD_PROCESS_SHARED set. In this case, *pthread_mutex_lock()* and *SyncMutexLock()* give an error of EINVAL.

Similarly, if you use `procnto`'s `-s` option, and you share a barrier or reader/writer lock between processes, you should use *pthread_barrierattr_setpshared()* or *pthread_rwlockattr_setpshared()* to set its process-shared attribute to PTHREAD_PROCESS_SHARED.

Semaphores

Semaphores are another common form of synchronization that allows threads to “post” and “wait” on a semaphore to control when threads wake or sleep.

The post (*sem_post()*) operation increments the semaphore; the wait (*sem_wait()*) operation decrements it.

If you wait on a semaphore that is positive, you will not block. Waiting on a nonpositive semaphore will block until some other thread executes a post. It is valid to post one or more times before a wait. This use will allow one or more threads to execute the wait without blocking.

A significant difference between semaphores and other synchronization primitives is that semaphores are “async safe” and can be manipulated by signal handlers. If the desired effect is to have a signal handler wake a thread, semaphores are the right choice.

Note that in general, mutexes are much faster than semaphores, which always require a kernel entry. Semaphores don't affect a thread's effective priority; if you need priority inheritance, use a mutex. For more information, see “*Mutexes: mutual exclusion locks*,” earlier in this chapter.

Another useful property of semaphores is that they were defined to operate between processes. Although our mutexes work between processes, the POSIX thread standard considers this an optional capability and as such may not be portable across systems. For synchronization between threads in a single process, mutexes will be more efficient than semaphores.

As a useful variation, a *named* semaphore service is also available. It lets you use semaphores between processes on different machines connected by a network.

Note that named semaphores are *slower* than the unnamed variety.

Since semaphores, like condition variables, can legally return a nonzero value because of a false wake-up, correct usage requires a loop:

```
while (sem_wait(&s) && (errno == EINTR)) { do_nothing(); }
do_critical_region();    /* Semaphore was decremented */
```

Synchronization via scheduling policy

By selecting the POSIX FIFO scheduling policy, we can guarantee that no two threads of the same priority execute the critical section concurrently on a non-SMP system.

The FIFO scheduling policy dictates that all FIFO-scheduled threads in the system at the same priority will run, when scheduled, until they voluntarily release the processor to another thread.

This “release” can also occur when the thread blocks as part of requesting the service of another process, or when a signal occurs. *The critical region must therefore be carefully coded and documented so that later maintenance of the code doesn't violate this condition.*

In addition, higher-priority threads in that (or any other) process could still preempt these FIFO-scheduled threads. So, all the threads that could “collide” within the critical section must be FIFO-scheduled at the *same* priority. Having enforced this condition, the threads can then casually access this shared memory without having to first make explicit synchronization calls.

This exclusive-access relationship doesn't apply in multiprocessor systems, since each CPU could run a thread simultaneously through the region that would otherwise be serially scheduled on a single-processor machine.

Synchronization via message passing

Our Send/Receive/Reply message-passing IPC services (described later) implement an implicit synchronization by their blocking nature. These IPC services can, in many instances, render other synchronization services unnecessary. They are also the only synchronization and IPC primitives (other than named semaphores, which are built on top of messaging) that can be used across the network.

Synchronization via atomic operations

In some cases, you may want to perform a short operation (such as incrementing a variable) with the guarantee that the operation will perform *atomically*—i.e., the operation won't be preempted by another thread or ISR (Interrupt Service Routine).

The QNX Neutrino RTOS provides atomic operations for:

- adding a value
- subtracting a value
- clearing bits
- setting bits
- toggling (complementing) bits

These atomic operations are available by including the C header file **<atomic.h>**.

Although you can use these atomic operations just about anywhere, you'll find them particularly useful in these two cases:

- between an ISR and a thread
- between two threads (SMP or single-processor)

Since an ISR can preempt a thread at any given point, the only way that the thread would be able to protect itself would be to *disable interrupts*. Since you should avoid disabling interrupts in a realtime system, we recommend that you use the atomic operations provided with QNX Neutrino.

On an SMP system, multiple threads *can* and *do* run concurrently. Again, we run into the same situation as with interrupts above—you should use the atomic operations where applicable to eliminate the need to disable and reenable interrupts.

Synchronization services implementation

The following table lists the various microkernel calls and the higher-level POSIX calls constructed from them:

Microkernel call	POSIX call	Description
SyncTypeCreate()	*pthread_mutex_init()*, *pthread_cond_init()*, *sem_init()*	Create object for mutex, condvars, and semaphore
SyncDestroy()	*pthread_mutex_destroy()*, *pthread_cond_destroy()*, *sem_destroy()*	Destroy synchronization object
SyncCondvarWait()	*pthread_cond_wait()*, *pthread_cond_timedwait()*	Block on a condvar
SyncCondvarSignal()	*pthread_cond_broadcast()*, *pthread_cond_signal()*	Wake up condvar-blocked threads
SyncMutexLock()	*pthread_mutex_lock()*, *pthread_mutex_trylock()*	Lock a mutex
SyncMutexUnlock()	*pthread_mutex_unlock()*	Unlock a mutex
SyncSemPost()	*sem_post()*	Post a semaphore
SyncSemWait()	*sem_wait()*, *sem_trywait()*	Wait on a semaphore

Clock and timer services

Clock services are used to maintain the time of day, which is in turn used by the kernel timer calls to implement interval timers.

Valid dates on a QNX Neutrino system range from January 1970 to at least 2038. The `time_t` data type is an unsigned 32-bit number, which extends this range for many applications through 2106. The kernel itself uses unsigned 64-bit numbers to count the nanoseconds since January 1970, and so can handle dates through 2554. If your system must operate past 2554 and there's no way for the system to be upgraded or modified in the field, you'll have to take special care with system dates (contact us for help with this).

The *ClockTime()* kernel call allows you to get or set the system clock specified by an ID (CLOCK_REALTIME), which maintains the system time. Once set, the system time increments by some number of nanoseconds based on the resolution of the system clock. This resolution can be queried or changed using the *ClockPeriod()* call.

Within the *system page*, an in-memory data structure, there's a 64-bit field (*nsec*) that holds the number of nanoseconds since the system was booted. The *nsec* field is always monotonically increasing and is never affected by setting the current time of day via *ClockTime()* or *ClockAdjust()*.

The *ClockCycles()* function returns the current value of a free-running 64-bit cycle counter. This is implemented on each processor as a high-performance mechanism for timing short intervals. For example, on Intel x86 processors, an opcode that reads the processor's time-stamp counter is used. Other CPU architectures have similar instructions.

On processors that don't implement such an instruction in hardware, the kernel will emulate one. This will provide a lower time resolution than if the instruction is provided (838.095345 nanoseconds on an IBM PC-compatible system).

In all cases, the `SYSPAGE_ENTRY(qtime)->cycles_per_sec` field gives the number of *ClockCycles()* increments in one second.

In QNX Neutrino 7.0 or later, we require that the hardware underlying *ClockCycles()* be synchronized across all processors on an SMP system. If it isn't, you might encounter some unexpected behavior, such as drifting times and timers.

The *ClockPeriod()* function allows a thread to set the system timer to some multiple of nanoseconds; the OS kernel will do the best it can to satisfy the precision of the request with the hardware available.

The interval selected is always rounded down to an integral of the precision of the underlying hardware timer. Of course, setting it to an extremely low value can result in a significant portion of CPU performance being consumed servicing timer interrupts.

The *ClockId()* function returns a special clock ID that you can use to track the CPU time that a process or thread uses. For more information, see “Monitoring execution times” in the “Understanding the Microkernel's Concept of Time” chapter of the QNX Neutrino *Programmer's Guide*.

Microkernel call	POSIX call	Description
ClockTime()	*clock_gettime()*, *clock_settime()*	Get or set the time of day (using a 64-bit value in nanoseconds ranging from 1970 to 2554)
ClockAdjust()	N/A	Apply small time adjustments to synchronize clocks.
ClockCycles()	N/A	Read a 64-bit free-running high-precision counter
ClockPeriod()	*clock_getres()*	Get or set the period of the clock
ClockId()	*clock_getcpuclockid()*, *pthread_getcpuclockid()*	Get a clock ID for a process or thread CPU-time clock

The kernel can run in a *tickless* mode in order to reduce power consumption, but this is a bit of a misnomer. The system still has clock ticks, and everything runs as normal unless the system is idle. Only when the system goes completely idle does the kernel turn off clock ticks, and in reality what it does is slow down the clock so that the next tick interrupt occurs just after the next active timer is to fire, so that the timer will fire immediately. To enable tickless operation, specify the `-Z` option for the `startup-*` code.

Time correction

In order to facilitate applying time corrections without having the system experience abrupt "steps" in time (or even having time jump backwards), the *ClockAdjust()* call provides the option to specify an interval over which the time correction is to be applied. This has the effect of speeding or retarding time over a specified interval until the system has synchronized to the indicated current time. This service can be used to implement network-coordinated time averaging between multiple nodes on a network.

Timers

The QNX Neutrino RTOS directly provides the full set of POSIX timer functionality. Since these timers are quick to create and manipulate, they're an inexpensive resource in the kernel.

The POSIX timer model is quite rich, providing the ability to have the timer expire on:

- an absolute date
- a relative date (i.e., *n* nanoseconds from now)
- cyclical (i.e., every *n* nanoseconds)

The cyclical mode is very significant, because the most common use of timers tends to be as a periodic source of events to "kick" a thread into life to do some processing and then go back to sleep until the next event. If the thread had to re-program the timer for every event, there would be the danger that time would slip unless the thread was programming an absolute date. Worse, if the thread doesn't get to run on the timer event because a higher-priority thread is running, the date next programmed into the timer could be one that has already elapsed!

The cyclical mode circumvents these problems by requiring that the thread set the timer once and then simply respond to the resulting periodic source of events.

Since timers are another source of events in the OS, they also make use of its event-delivery system. As a result, the application can request that any of the QNX Neutrino-supported events be delivered to the application upon occurrence of a timeout.

An often-needed timeout service provided by the OS is the ability to specify the maximum time the application is prepared to wait for any given kernel call or request to complete. A problem with using generic OS timer services in a preemptive realtime OS is that in the interval between the specification of the timeout and the request for the service, a higher-priority process might have been scheduled to run and preempted long enough that the specified timeout will have expired before the service is even requested. The application will then end up requesting the service with an already lapsed timeout in effect (i.e., no timeout). This timing window can result in “hung” processes, inexplicable delays in data transmission protocols, and other problems.

```
alarm(...);
    ...
    ...    ← Alarm fires here
    ...
blocking_call();
```

Our solution is a form of timeout request atomic to the service request itself. One approach might have been to provide an optional timeout parameter on every available service request, but this would overly complicate service requests with a passed parameter that would often go unused.

QNX Neutrino provides a *TimerTimeout()* kernel call that allows an application to specify a list of blocking states for which to start a specified timeout. Later, when the application makes a request of the kernel, the kernel will atomically enable the previously configured timeout if the application is about to block on one of the specified states.

Since the OS has a very small number of blocking states, this mechanism works very concisely. At the conclusion of either the service request or the timeout, the timer will be disabled and control will be given back to the application.

```
TimerTimeout(...);
    ...
    ...
    ...
blocking_call();
    ...    ← Timer atomically armed within kernel
```

Microkernel call	POSIX call	Description
TimerAlarm()	*alarm()*	Set a process alarm
TimerCreate()	*timer_create()*	Create an interval timer
TimerDestroy()	*timer_delete()*	Destroy an interval timer
TimerInfo()	*timer_gettime()*	Get the time remaining on an interval timer

Microkernel call	**POSIX call**	**Description**
TimerInfo()	*timer_getoverrun()*	Get the number of overruns on an interval timer
TimerSettime()	*timer_settime()*	Start an interval timer
TimerTimeout()	*sleep()*, *nanosleep()*, *sigtimedwait()*, *pthread_cond_timedwait()*, *pthread_mutex_trylock()*	Arm a kernel timeout for any blocking state

For more information, see the Clocks, Timers, and Getting a Kick Every So Often chapter of *Getting Started with QNX Neutrino*.

Interrupt handling

No matter how much we wish it were so, computers are not infinitely fast. In a realtime system, it's absolutely crucial that CPU cycles aren't unnecessarily spent. It's also crucial to minimize the time from the occurrence of an external event to the actual execution of code within the thread responsible for reacting to that event. This time is referred to as *latency*.

The two forms of latency that most concern us are interrupt latency and scheduling latency.

Latency times can vary significantly, depending on the speed of the processor and other factors. For more information, visit our website (*www.qnx.com*).

Interrupt latency

Interrupt latency is the time from the assertion of a hardware interrupt until the first instruction of the device driver's interrupt handler is executed.

The OS leaves interrupts fully enabled almost all the time, so that interrupt latency is typically insignificant. But certain critical sections of code do require that interrupts be temporarily disabled. The maximum such disable time usually defines the worst-case interrupt latency—in QNX Neutrino this is very small.

The following diagrams illustrate the case where a hardware interrupt is processed by an established interrupt handler. The interrupt handler either will simply return, or it will return and cause an event to be delivered.

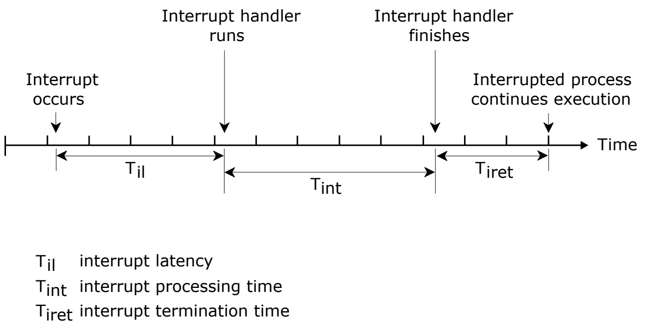

Interrupt handler runs
Interrupt handler finishes
Interrupt occurs
Interrupted process continues execution
Time
T_{il}
T_{int}
T_{iret}
T_{il} interrupt latency
T_{int} interrupt processing time
T_{iret} interrupt termination time

Figure 15: Interrupt handler simply terminates.

The interrupt latency (T*il*) in the above diagram represents the *minimum* latency—that which occurs when interrupts were fully enabled at the time the interrupt occurred. Worst-case interrupt latency will be this time *plus* the longest time in which the OS, or the running system process, disables CPU interrupts.

Scheduling latency

In some cases, the low-level hardware interrupt handler must schedule a higher-level thread to run. In this scenario, the interrupt handler will return and indicate that an event is to be delivered. This introduces a second form of latency—*scheduling latency*—which must be accounted for.

Scheduling latency is the time between the last instruction of the user's interrupt handler and the execution of the first instruction of a driver thread. This usually means the time it takes to save the context of the currently executing thread and restore the context of the required driver thread. Although larger than interrupt latency, this time is also kept small in a QNX Neutrino system.

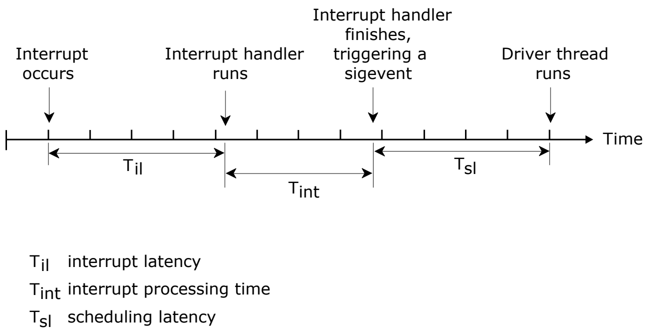

Interrupt occurs
Interrupt handler runs
Interrupt handler finishes, triggering a sigevent
Driver thread runs
Time
T_{il}
T_{int}
T_{sl}
T_{il} interrupt latency
T_{int} interrupt processing time
T_{sl} scheduling latency

Figure 16: Interrupt handler terminates, returning an event.

It's important to note that *most* interrupts terminate without delivering an event. In a large number of cases, the interrupt handler can take care of all hardware-related issues. Delivering an event to wake up a higher-level *driver* thread occurs only when a significant event occurs. For example, the interrupt handler for a serial device driver would feed one byte of data to the hardware upon each received transmit interrupt, and would trigger the higher-level thread within (`devc-ser*`) only when the output buffer is nearly empty.

Nested interrupts

The QNX Neutrino RTOS fully supports nested interrupts.

The previous scenarios describe the simplest—and most common—situation where only one interrupt occurs. Worst-case timing considerations for unmasked interrupts must take into account the time for all interrupts currently being processed, because a higher priority, unmasked interrupt will preempt an existing interrupt.

In the following diagram, Thread A is running. Interrupt IRQ_x causes interrupt handler Int_x to run, which is preempted by IRQ_y and its handler Int_y. Int_y returns an event causing Thread B to run; Int_x returns an event causing Thread C to run.

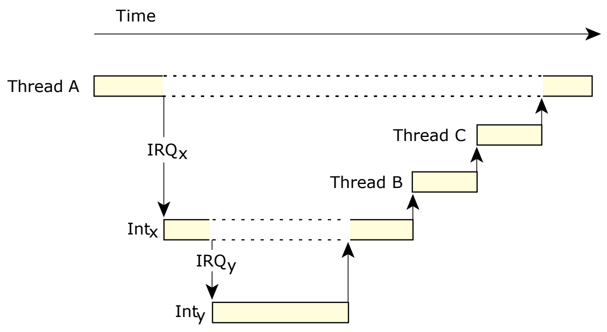


Figure 17: Stacked interrupts.

Interrupt calls

The interrupt-handling API includes the following kernel calls:

Function	Description
InterruptAttach()	Attach a local function (an Interrupt Service Routine or ISR) to an interrupt vector.
InterruptAttachEvent()	Generate an event on an interrupt, which will ready a thread. No user interrupt handler runs. This is the preferred call.
InterruptDetach()	Detach from an interrupt using the ID returned by *InterruptAttach()* or *InterruptAttachEvent()*.
InterruptWait()	Wait for an interrupt.
InterruptEnable()	Enable hardware interrupts.
InterruptDisable()	Disable hardware interrupts.
InterruptMask()	Mask a hardware interrupt.
InterruptUnmask()	Unmask a hardware interrupt.
InterruptLock()	Guard a critical section of code between an interrupt handler and a thread. A spinlock is used to make this code SMP-safe. This function is a superset of *InterruptDisable()* and should be used in its place.
InterruptUnlock()	Remove an SMP-safe lock on a critical section of code.

Using this API, a suitably privileged user-level thread can call *InterruptAttach()* or *InterruptAttachEvent()*, passing a hardware interrupt number and the address of a function in the thread's address space to be called when the interrupt occurs. QNX Neutrino allows multiple ISRs to be attached to each hardware interrupt number—unmasked interrupts can be serviced during the execution of running interrupt handlers.

- The startup code is responsible for making sure that all interrupt sources are masked during system initialization. When the first call to *InterruptAttach()* or *InterruptAttachEvent()* is done for an interrupt vector, the kernel unmasks it. Similarly, when the last *InterruptDetach()* is done for an interrupt vector, the kernel remasks the level.
- For more information on *InterruptLock()* and *InterruptUnlock()*, see "*Critical sections*" in the chapter on Multicore Processing in this guide.
- It isn't safe to use floating-point operations in Interrupt Service Routines.

The following code sample shows how to attach an ISR to the hardware timer interrupt on the PC (which the OS also uses for the system clock). Since the kernel's timer ISR is already dealing with clearing the source of the interrupt, this ISR can simply increment a counter variable in the thread's data space and return to the kernel:

```
#include <stdio.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>

struct sigevent event;
volatile unsigned counter;

const struct sigevent *handler( void *area, int id ) {
    // Wake up the thread every 100th interrupt
    if ( ++counter == 100 ) {
        counter = 0;
        return( &event );
        }
    else
        return( NULL );
    }

int main() {
    int i;
    int id;

    // Initialize event structure
    event.sigev_notify = SIGEV_INTR;

    // Enable the INTERRUPT ability
    procmgr_ability(0,
        PROCMGR_ADN_ROOT|PROCMGR_AOP_ALLOW|PROCMGR_AID_INTERRUPT,
        PROCMGR_AID_EOL);

    // Attach ISR vector
    id=InterruptAttach( SYSPAGE_ENTRY(qtime)->intr, &handler,
                        NULL, 0, 0 );

    for( i = 0; i < 10; ++i ) {
        // Wait for ISR to wake us up
        InterruptWait( 0, NULL );
        printf( "100 events\n" );
        }
    // Disconnect the ISR handler
    InterruptDetach(id);
    return 0;
    }
```

With this approach, appropriately privileged user-level threads can dynamically attach (and detach) interrupt handlers to (and from) hardware interrupt vectors at run time. These threads can be debugged using regular source-level debug tools; the ISR itself can be debugged by calling it at the thread level and source-level stepping through it or by using the *InterruptAttachEvent()* call.

When the hardware interrupt occurs, the processor will enter the interrupt redirector in the microkernel. This code pushes the registers for the context of the currently running thread into the appropriate thread table entry and sets the processor context such that the ISR has access to the code and data that are part of the thread the ISR is contained within. This allows the ISR to use the buffers and code in the user-level thread to resolve the interrupt and, if higher-level work by the thread is required, to queue an event to the thread the ISR is part of, which can then work on the data the ISR has placed into thread-owned buffers.

Since it runs with the memory-mapping of the thread containing it, the ISR can directly manipulate devices mapped into the thread's address space, or directly perform I/O instructions. As a result, device drivers that manipulate hardware don't need to be linked into the kernel.

The interrupt redirector code in the microkernel will call each ISR attached to that hardware interrupt. If the value returned indicates that a process is to be passed an event of some sort, the kernel will queue the event. When the last ISR has been called for that vector, the kernel interrupt handler will finish manipulating the interrupt control hardware and then "return from interrupt."

This interrupt return won't necessarily be into the context of the thread that was interrupted. If the queued event caused a higher-priority thread to become READY, the microkernel will then interrupt-return into the context of the now-READY thread instead.

This approach provides a well-bounded interval from the occurrence of the interrupt to the execution of the first instruction of the user-level ISR (measured as *interrupt latency*), and from the last instruction of the ISR to the first instruction of the thread readied by the ISR (measured as thread or process *scheduling latency*).

The worst-case interrupt latency is well-bounded, because the OS disables interrupts only for a couple opcodes in a few critical regions. Those intervals when interrupts are disabled have deterministic runtimes, because they're not data dependent.

The microkernel's interrupt redirector executes only a few instructions before calling the user's ISR. As a result, process preemption for hardware interrupts or kernel calls is equally quick and exercises essentially the same code path.

While the ISR is executing, it has full hardware access (since it's part of a privileged thread), but can't issue other kernel calls. The ISR is intended to respond to the hardware interrupt in as few microseconds as possible, do the minimum amount of work to satisfy the interrupt (read the byte from the UART, etc.), and if necessary, cause a thread to be scheduled at some user-specified priority to do further work.

Worst-case interrupt latency is directly computable for a given hardware priority from the kernel-imposed interrupt latency and the maximum ISR runtime for each interrupt higher in hardware priority than the

ISR in question. Since hardware interrupt priorities can be reassigned, the most important interrupt in the system can be made the highest priority.

Note also that by using the *InterruptAttachEvent()* call, no user ISR is run. Instead, a user-specified event is generated on each and every interrupt; the event will typically cause a waiting thread to be scheduled to run and do the work. The interrupt is automatically masked when the event is generated and then explicitly unmasked by the thread that handles the device at the appropriate time.

Both *InterruptMask()* and *InterruptUnmask()* are *counting* functions. For example, if *InterruptMask()* is called ten times, then *InterruptUnmask()* must also be called ten times.

Thus the priority of the work generated by hardware interrupts can be performed at OS-scheduled priorities rather than hardware-defined priorities. Since the interrupt source won't re-interrupt until serviced, the effect of interrupts on the runtime of critical code regions for hard-deadline scheduling can be controlled.

In addition to hardware interrupts, various “events” within the microkernel can also be “hooked” by user processes and threads. When one of these events occurs, the kernel can upcall into the indicated function in the user thread to perform some specific processing for this event. For example, whenever the idle thread in the system is called, a user thread can have the kernel upcall into the thread so that hardware-specific low-power modes can be readily implemented.

Microkernel call	Description
InterruptHookIdle2()	When the kernel has no active thread to schedule, it runs the idle thread, which can call a user handler. This handler can perform hardware-specific power-management operations.
InterruptHookTrace()	This function attaches a pseudo interrupt handler that can receive trace events from the instrumented kernel.

For more information about interrupts, see the Interrupts chapter of *Getting Started with QNX Neutrino*, and the Writing an Interrupt Handler chapter of the QNX Neutrino *Programmer's Guide*.

Chapter 3 Interprocess Communication (IPC)

Interprocess Communication plays a fundamental role in the transformation of the microkernel from an embedded realtime kernel into a full-scale POSIX operating system. As various service-providing processes are added to the microkernel, IPC is the “glue” that connects those components into a cohesive whole.

Although message passing is the primary form of IPC in the QNX Neutrino RTOS, several other forms are available as well. Unless otherwise noted, those other forms of IPC are built over our native message passing. The strategy is to create a simple, robust IPC service that can be tuned for performance through a simplified code path in the microkernel; more “feature cluttered” IPC services can then be implemented from these.

Benchmarks comparing higher-level IPC services (like pipes and FIFOs implemented over our messaging) with their monolithic kernel counterparts show comparable performance.

QNX Neutrino offers at least the following forms of IPC:

Service:	Implemented in:
Message-passing	Kernel
Signals	Kernel
POSIX message queues	External process
Shared memory	Process manager
Pipes	External process
FIFOs	External process

The designer can select these services on the basis of bandwidth requirements, the need for queuing, network transparency, etc. The trade-off can be complex, but the flexibility is useful.

As part of the engineering effort that went into defining the microkernel, the focus on message passing as the fundamental IPC primitive was deliberate. As a form of IPC, message passing (as implemented in *MsgSend()*, *MsgReceive()*, and *MsgReply()*), is synchronous and copies data. Let's explore these two attributes in more detail.

Synchronous message passing

Synchronous messaging is the main form of IPC in the QNX Neutrino RTOS.

A thread that does a *MsgSend()* to another thread (which could be within another process) will be blocked until the target thread does a *MsgReceive()*, processes the message, and executes a *MsgReply()*. If a thread executes a *MsgReceive()* without a previously sent message pending, it will block until another thread executes a *MsgSend()*.

In QNX Neutrino, a server thread typically loops, waiting to receive a message from a client thread. As described earlier, a thread—whether a server or a client—is in the READY state if it can use the CPU. It might not actually be getting any CPU time because of its and other threads' priority and scheduling policy, but the thread isn't blocked.

Let's look first at the client thread:

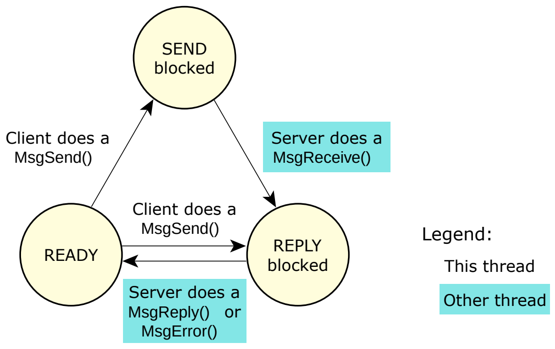


Figure 18: Changes of state for a client thread in a send-receive-reply transaction.

- If the client thread calls *MsgSend()*, and the server thread hasn't yet called *MsgReceive()*, then the client thread becomes SEND blocked. Once the server thread calls *MsgReceive()*, the kernel changes the client thread's state to be REPLY blocked, which means that server thread has received the message and now must reply. When the server thread calls *MsgReply()*, the client thread becomes READY.
- If the client thread calls *MsgSend()*, and the server thread is already blocked on the *MsgReceive()*, then the client thread immediately becomes REPLY blocked, skipping the SEND-blocked state completely.
- If the server thread fails, exits, or disappears, the client thread becomes READY, with *MsgSend()* indicating an error.

Next, let's consider the server thread:

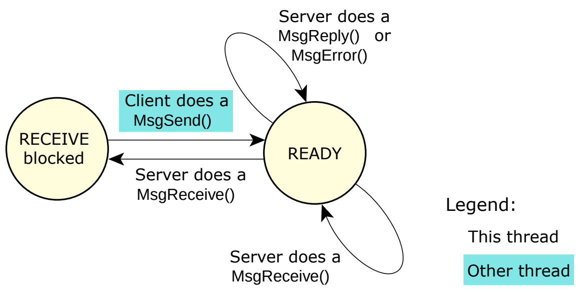

Server does a MsgReply() or MsgError()
Client does a MsgSend()
RECEIVE blocked
READY
Server does a MsgReceive()
Server does a MsgReceive()
Legend:
This thread
Other thread

Figure 19: Changes of state for a server thread in a send-receive-reply transaction.

- If the server thread calls *MsgReceive()*, and no other thread has sent to it, then the server thread becomes RECEIVE blocked. When another thread sends to it, the server thread becomes READY.
- If the server thread calls *MsgReceive()*, and another thread has already sent to it, then *MsgReceive()* returns immediately with the message. In this case, the server thread doesn't block.
- If the server thread calls *MsgReply()*, it doesn't become blocked.

This inherent blocking synchronizes the execution of the sending thread, since the act of requesting that the data be sent also causes the sending thread to be blocked and the receiving thread to be scheduled for execution. This happens without requiring explicit work by the kernel to determine which thread to run next (as would be the case with most other forms of IPC). Execution and data move directly from one context to another.

Data-queuing capabilities are omitted from these messaging primitives because queueing could be implemented when needed within the receiving thread. The sending thread is often prepared to wait for a response; queueing is unnecessary overhead and complexity (i.e., it slows down the nonqueued case). As a result, the sending thread doesn't need to make a separate, explicit blocking call to wait for a response (as it would if some other IPC form had been used).

While the send and receive operations are blocking and synchronous, *MsgReply()* (or *MsgError()*) doesn't block. Since the client thread is already blocked waiting for the reply, no additional synchronization is required, so a blocking *MsgReply()* isn't needed. This allows a server to reply to a client and continue processing while the kernel and/or networking code asynchronously passes the reply data to the sending thread and marks it ready for execution. Since most servers will tend to do some processing to prepare to receive the next request (at which point they block again), this works out well.

Note that in a network, a reply may not complete as “immediately” as in a local message pass. For more information on network message passing, see the chapter on *Qnet networking* in this book.

MsgReply()* vs *MsgError()

The *MsgReply()* function is used to return a status and zero or more bytes to the client. *MsgError()*, on the other hand, is used to return *only* a status to the client. Both functions will unblock the client from its *MsgSend()*.

Message copying

Since our messaging services copy a message directly from the address space of one thread to another without intermediate buffering, the message-delivery performance approaches the memory bandwidth of the underlying hardware.

The kernel attaches no special meaning to the content of a message—the data in a message has meaning only as mutually defined by sender and receiver. However, “well-defined” message types are also provided so that user-written processes or threads can augment or substitute for system-supplied services.

The messaging primitives support multipart transfers, so that a message delivered from the address space of one thread to another needn't pre-exist in a single, contiguous buffer. Instead, both the sending and receiving threads can specify a vector table that indicates where the sending and receiving message fragments reside in memory. Note that the size of the various parts can be different for the sender and receiver.

Multipart transfers allow messages that have a header block separate from the data block to be sent without performance-consuming copying of the data to create a contiguous message. In addition, if the underlying data structure is a ring buffer, specifying a three-part message will allow a header and two disjoint ranges within the ring buffer to be sent as a single atomic message. A hardware equivalent of this concept would be that of a scatter/gather DMA facility.

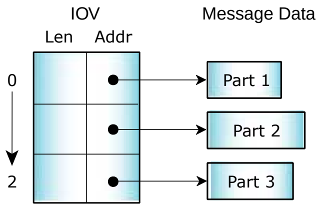


Figure 20: A multipart transfer.

Each IOV can have a maximum of 524288 parts. The sum of the sizes of the parts must not exceed INT_MAX.

The multipart transfers are also used extensively by filesystems. On a read, the data is copied directly from the filesystem cache into the application using a message with *n* parts for the data. Each part points into the cache and compensates for the fact that cache blocks aren't contiguous in memory with a read starting or ending within a block.

For example, with a cache block size of 512 bytes, a read of 1454 bytes can be satisfied with a five-part message:

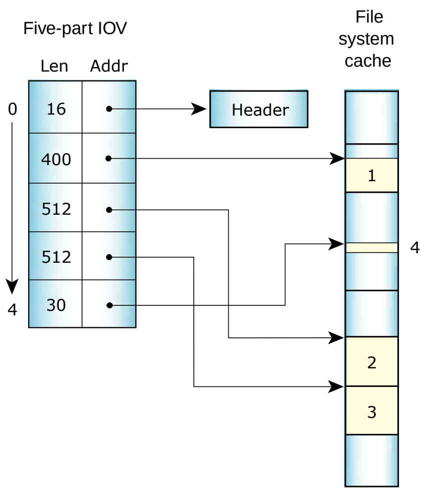


Figure 21: Scatter/gather of a read of 1454 bytes.

Since message data is explicitly copied between address spaces (rather than by doing page table manipulations), messages can be easily allocated on the stack instead of from a special pool of page-aligned memory for MMU "page flipping." As a result, many of the library routines that implement the API between client and server processes can be trivially expressed, without elaborate IPC-specific memory allocation calls.

For example, the code used by a client thread to request that the filesystem manager execute `lseek` on its behalf is implemented as follows:

```
#include <unistd.h>
#include <errno.h>
#include <sys/iomsg.h>

off64_t lseek64(int fd, off64_t offset, int whence) {
    io_lseek_t                          msg;
    off64_t                             off;

    msg.i.type = _IO_LSEEK;
    msg.i.combine_len = sizeof msg.i;
    msg.i.offset = offset;
    msg.i.whence = whence;
    msg.i.zero = 0;
    if(MsgSend(fd, &msg.i, sizeof msg.i, &off, sizeof off) == -1) {
        return -1;
    }
    return off;
}

off_t lseek(int fd, off_t offset, int whence) {
    return lseek64(fd, offset, whence);
}
```

This code essentially builds a message structure on the stack, populates it with various constants and passed parameters from the calling thread, and sends it to the filesystem manager associated with *fd*. The reply indicates the success or failure of the operation.

This implementation doesn't prevent the kernel from detecting large message transfers and choosing to implement “page flipping” for those cases. Since most messages passed are quite tiny, copying messages is often faster than manipulating MMU page tables. For bulk data transfer, shared memory between processes (with message-passing or the other synchronization primitives for notification) is also a viable option.

Simple messages

For simple single-part messages, the OS provides functions that take a pointer directly to a buffer without the need for an IOV (input/output vector). In this case, the number of parts is replaced by the size of the message directly pointed to.

In the case of the *message send* primitive—which takes a send and a reply buffer—this introduces four variations:

Function	Send message	Reply message
MsgSend()	Simple	Simple
MsgSendsv()	Simple	IOV
MsgSendvs()	IOV	Simple
MsgSendv()	IOV	IOV

The other messaging primitives that take a direct message simply drop the trailing “v” in their names:

IOV	Simple direct
MsgReceivev()	*MsgReceive()*
MsgReceivePulsev()	*MsgReceivePulse()*
MsgReplyv()	*MsgReply()*
MsgReadv()	*MsgRead()*
MsgWritev()	*MsgWrite()*

Channels and connections

In the QNX Neutrino RTOS, message passing is directed towards channels and connections, rather than targeted directly from thread to thread. A thread that wishes to receive messages first creates a channel; another thread that wishes to send a message to that thread must first make a connection by “attaching” to that channel.

Channels are required by the message kernel calls and are used by servers to *MsgReceive()* messages on. Connections are created by client threads to “connect” to the channels made available by servers. Once connections are established, clients can *MsgSend()* messages over them. If a number of threads in a process all attach to the same channel, then the connections all map to the same kernel object for efficiency. Channels and connections are named within a process by a small integer identifier. Client connections map directly into file descriptors.

Architecturally, this is a key point. By having client connections map directly into FDs, we have eliminated yet another layer of translation. We don't need to “figure out” where to send a message based on the file descriptor (e.g., via a `read(fd)` call). Instead, we can simply send a message directly to the “file descriptor” (i.e., connection ID).

Function	Description
ChannelCreate()	Create a channel to receive messages on.
ChannelDestroy()	Destroy a channel.
ConnectAttach()	Create a connection to send messages on.
ConnectDetach()	Detach a connection.

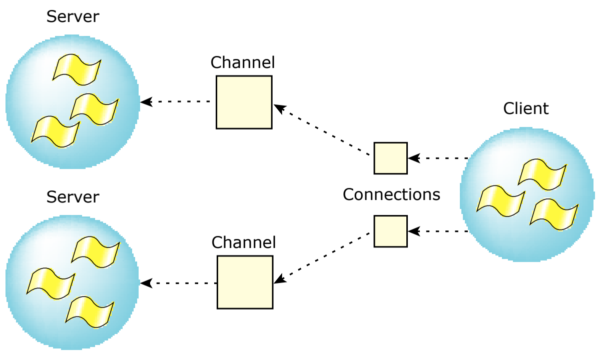


Figure 22: Connections map elegantly into file descriptors.

A process acting as a server would implement an event loop to receive and process messages as follows:

```
chid = ChannelCreate(flags);
SETIOV(&iov, &msg, sizeof(msg));
for(;;) {
    rcv_id = MsgReceivev( chid, &iov, parts, &info );

    switch( msg.type ) {
        /* Perform message processing here */
        }

    MsgReplyv( rcv_id, &iov, rparts );
    }
```

This loop allows the thread to receive messages from any thread that had a connection to the channel.

The server can also use *name_attach()* to create a channel and associate a name with it. The sender process can then use *name_open()* to locate that name and create a connection to it.

The channel has several lists of messages associated with it:

Receive

A LIFO queue of threads waiting for messages.

Send

A priority FIFO queue of threads that have sent messages that haven't yet been received.

Reply

An unordered list of threads that have sent messages that have been received, but not yet replied to.

While in any of these lists, the waiting thread is blocked (i.e., RECEIVE-, SEND-, or REPLY-blocked). Multiple threads and multiple clients may wait on one channel.

Pulses

In addition to the synchronous Send/Receive/Reply services, the OS also supports fixed-size, nonblocking messages. These are referred to as *pulses* and carry a small payload (four bytes of data plus a single byte code).

Pulses pack a relatively small payload—eight bits of code and 32 bits of data. Pulses are often used as a notification mechanism within interrupt handlers. They also allow servers to signal clients without blocking on them.

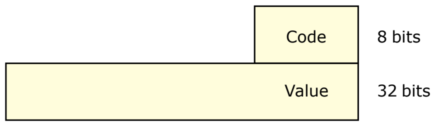


Figure 23: Pulses pack a small payload.

Priority inheritance and messages

A server process receives messages and pulses in priority order. As the threads within the server receive requests, they then inherit the priority (but not the scheduling policy) of the sending thread. As a result, the relative priorities of the threads requesting work of the server are preserved, and the server work will be executed at the appropriate priority. This message-driven priority inheritance avoids priority-inversion problems.

For example, suppose the system includes the following:

- a server thread, at priority 22
- a client thread, T1, at priority 13
- a client thread, T2, at priority 10

Without priority inheritance, if T2 sends a message to the server, it's effectively getting work done for it at priority 22, so T2's priority has been inverted.

What actually happens is that when the server receives a message, its effective priority changes to that of the highest-priority sender (restricted as described below). In this case, T2's priority is lower than the server's, so the change in the server's effective priority takes place when the server *receives* the message.

Next, suppose that T1 sends a message to the server while it's still at priority 10. Since T1's priority is higher than the server's current priority, the change in the server's priority happens when T1 *sends* the message.

The change happens before the server receives the message to avoid another case of priority inversion. If the server's priority remains unchanged at 10, and another thread, T3, starts to run at priority 11, the server has to wait until T3 lets it have some CPU time so that it can eventually receive T1's message. So, T1 would be delayed by a lower-priority thread, T3.

If the highest priority among the threads is a privileged priority, and the server doesn't have the PROCMGR_AID_PRIORITY ability enabled, then the server thread is boosted to the highest unprivileged priority. For more information, see "*Scheduling priority*" in the "QNX Neutrino Microkernel" chapter and *procmgr_ability()* in the *C Library Reference*.

You can turn off priority inheritance by specifying the _NTO_CHF_FIXED_PRIORITY flag when you call *ChannelCreate()*. If you're using adaptive partitioning, this flag also causes the receiving threads not to run in the sending threads' partitions.

Server boost

If a high-priority thread becomes SEND-blocked on a server, the kernel tries to find one or more threads that are likely to receive on the given channel, and then it boosts their priorities.

When there are no server threads that are RECEIVE-blocked on a channel, and a client sends a message or pulse, the kernel boosts the priority of all server threads that last received on that channel, in the hope that one of them will soon finish up and be able to do a receive.

This situation is considered to be a symptom of a poorly designed server, and the kernel's response is an attempt to work around it. A server or resource manager should always have at least one RECEIVE-blocked thread.

When a thread is boosted for the first time, its original priority is recorded. The thread may be boosted multiple times if multiple threads become SEND-blocked, but the kernel records only the priority before the initial boosting.

When a message is next received, the kernel reevaluates the situation. If there are still pulses or SEND-blocked threads, then some of the boosted threads may remain boosted (although potentially at a lower priority), while others may drop back to their original priority.

If there are no SEND-blocked threads, then all threads that were originally boosted return to their original priorities.

A thread is associated with a channel when it does a *MsgReceive()* on it. In the case of multiple channels, a thread is associated with the last channel it received from. After receiving a message, a thread can dissociate itself from the channel by calling *MsgReceive()* with a -1 for the channel ID.

Message-passing API

The message-passing API consists of the following functions:

Function	Description
MsgSend()	Send a message and block until reply.
MsgReceive()	Wait for a message.
MsgReceivePulse()	Wait for a tiny, nonblocking message (pulse).
MsgReply()	Reply to a message.
MsgError()	Reply only with an error status. No message bytes are transferred.
MsgRead()	Read additional data from a received message.
MsgWrite()	Write additional data to a reply message.
MsgInfo()	Obtain info on a received message.
MsgSendPulse()	Send a tiny, nonblocking message (pulse).
MsgDeliverEvent()	Deliver an event to a client.
MsgKeyData()	Key a message to allow security checks.

For information about messages from the programming point of view, see the Message Passing chapter of *Getting Started with QNX Neutrino*.

Robust implementations with Send/Receive/Reply

Architecting a QNX Neutrino application as a team of cooperating threads and processes via Send/Receive/Reply results in a system that uses *synchronous* notification. IPC thus occurs at specified transitions within the system, rather than asynchronously.

A significant problem with asynchronous systems is that event notification requires signal handlers to be run. Asynchronous IPC can make it difficult to thoroughly test the operation of the system and make sure that no matter when the signal handler runs, that processing will continue as intended. Applications often try to avoid this scenario by relying on a "window" explicitly opened and shut, during which signals will be tolerated.

With a synchronous, nonqueued system architecture built around Send/Receive/Reply, robust application architectures can be very readily implemented and delivered.

Avoiding deadlock situations is another difficult problem when constructing applications from various combinations of queued IPC, shared memory, and miscellaneous synchronization primitives. For example, suppose thread A doesn't release mutex 1 until thread B releases mutex 2. Unfortunately, if thread B is in the state of not releasing mutex 2 until thread A releases mutex 1, a standoff results. Simulation tools are often invoked in order to ensure that deadlock won't occur as the system runs.

The Send/Receive/Reply IPC primitives allow the construction of deadlock-free systems with the observation of only these simple rules:

1. Never have two threads send to each other.
2. Always arrange your threads in a hierarchy, with sends going up the tree.

The first rule is an obvious avoidance of the standoff situation, but the second rule requires further explanation. The team of cooperating threads and processes is arranged as follows:

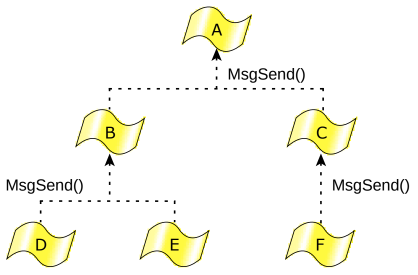


Figure 24: Threads should always send up to higher-level threads.

Here the threads at any given level in the hierarchy never send to each other, but send only upwards instead.

One example of this might be a client application that sends to a database server process, which in turn sends to a filesystem process. Since the sending threads block and wait for the target thread to reply, and since the target thread isn't send-blocked on the sending thread, deadlock can't happen.

But how does a higher-level thread notify a lower-level thread that it has the results of a previously requested operation? (Assume the lower-level thread didn't want to wait for the replied results when it last sent.)

The QNX Neutrino RTOS provides a very flexible architecture with the *MsgDeliverEvent()* kernel call to deliver nonblocking events. All of the common asynchronous services can be implemented with this.

For example, the server-side of the *select()* call is an API that an application can use to allow a thread to wait for an I/O event to complete on a set of file descriptors. In addition to an asynchronous notification mechanism being needed as a "back channel" for notifications from higher-level threads to lower-level threads, we can also build a reliable notification system for timers, hardware interrupts, and other event sources around this.

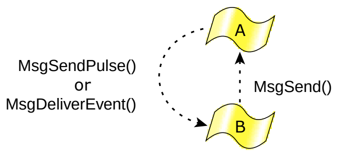


Figure 25: A higher-level thread can "send" a pulse event.

A related issue is the problem of how a higher-level thread can request work of a lower-level thread without sending to it, risking deadlock. The lower-level thread is present only to serve as a "worker thread" for the higher-level thread, doing work on request. The lower-level thread would send in order to "report for work," but the higher-level thread wouldn't reply then. It would defer the reply until the higher-level thread had work to be done, and it would reply (which is a nonblocking operation) with the data describing the work. In effect, the reply is being used to initiate work, not the send, which neatly side-steps rule #1.

Events

A significant feature in the kernel design for QNX Neutrino is the event-handling subsystem. POSIX and its realtime extensions define a number of asynchronous notification methods (e.g., UNIX signals that don't queue or pass data, POSIX realtime signals that may queue and pass data, etc.).

The kernel also defines additional, QNX Neutrino-specific notification techniques such as pulses. Implementing all of these event mechanisms could have consumed significant code space, so our implementation strategy was to build all of these notification methods over a single, rich, event subsystem.

A benefit of this approach is that capabilities exclusive to one notification technique can become available to others. For example, an application can apply the same queueing services of POSIX realtime signals to UNIX signals. This can simplify the robust implementation of signal handlers within applications.

The events encountered by an executing thread can come from any of three sources:

- a *MsgDeliverEvent()* kernel call invoked by a thread
- an interrupt handler
- the expiry of a timer

The event itself can be any of a number of different types: QNX Neutrino pulses, interrupts, various forms of signals, and forced “unblock” events. “Unblock” is a means by which a thread can be released from a deliberately blocked state without any explicit event actually being delivered.

Given this multiplicity of event types, and applications needing the ability to request whichever asynchronous notification technique best suits their needs, it would be awkward to require that server processes (the higher-level threads from the previous section) carry code to support all these options.

Instead, the client thread can give a data structure, or “cookie,” called a `sigevent` to the server to hang on to until later. When the server needs to notify the client thread, it invokes *MsgDeliverEvent()*, and the microkernel sets the event type encoded within the cookie upon the client thread.

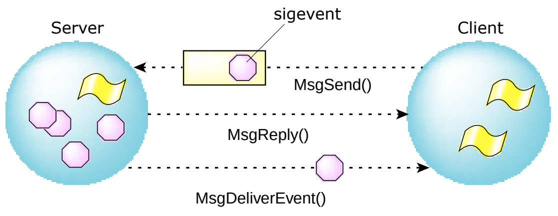


Figure 26: The client sends a `sigevent` to the server.

For details about the `sigevent` structure, see the *C Library Reference*; for a description of how to use it, see “Notification schemes” in the “Clocks, Timers, and Getting a Kick Every So Often” chapter of *Getting Started with QNX Neutrino*.

In a deeply embedded system that consists entirely of trusted programs, no safeguards are needed on events, but this isn't the case in systems that are more open. One problem with `sigevent`s is that a server can deliver any event to a client, even if the client has never expressed an interest in receiving events. In QNX Neutrino 7.0.1 or later, the client can register events to make them more secure, as follows:

1. The client sets up the `sigevent` to indicate how it wants to be notified.
2. If the client wants to allow the server to update the value associated with the event, it sets the SIGEV_FLAG_UPDATEABLE flag in the event.
3. The client registers an event by calling *MsgRegisterEvent()*. This function registers the given `sigevent` with the kernel, which stores the event internally and provides a handle for the event.
4. To prevent the delivery of unregistered events, the client sets the _NTO_COF_REG_EVENTS flag when it calls *ConnectAttach()* to connect to the server.
5. The client gives the handle instead of the actual event to the server.
6. If the client set the SIGEV_FLAG_UPDATEABLE flag on the event, then the server is allowed to update the value included in the registered event.
7. When the server calls *MsgDeliverEvent()* with a handle, the kernel looks up the handle. If the handle exists and the client has allowed the server to deliver it, the kernel delivers the corresponding registered event to the client.
8. When the client no longer needs the secure event, it can call *MsgUnregisterEvent()* to remove it and unregister the handle.

The client can thus be sure that it gets only the events that it wants, and that no one has tampered with them. For a sample program, see the entry for *MsgRegisterEvent()* in the *C Library Reference*.

I/O notification

The *ionotify()* function is a means by which a client thread can request asynchronous event delivery.

Many of the POSIX asynchronous services (e.g., *mq_notify()* and the client-side of the *select()*) are built on top of *ionotify()*. When performing I/O on a file descriptor (*fd*), the thread may choose to wait for an I/O event to complete (for the *write()* case), or for data to arrive (for the *read()* case). Rather than have the thread block on the resource manager process that's servicing the read/write request, *ionotify()* can allow the client thread to post an event to the resource manager that the client thread would like to receive when the indicated I/O condition occurs. Waiting in this manner allows the thread to continue executing and responding to event sources other than just the single I/O request.

The *select()* call is implemented using I/O notification and allows a thread to block and wait for a mix of I/O events on multiple *fd*'s while continuing to respond to other forms of IPC.

Here are the conditions upon which the requested event can be delivered:

- _NOTIFY_COND_OUTPUT—there's room in the output buffer for more data.
- _NOTIFY_COND_INPUT—resource-manager-defined amount of data is available to read.
- _NOTIFY_COND_OBAND—resource-manager-defined “out of band” data is available.

Signals

The OS supports the 32 standard POSIX signals (as in UNIX) as well as the POSIX realtime signals, both numbered from a kernel-implemented set of 64 signals with uniform functionality. While the POSIX standard defines realtime signals as differing from UNIX-style signals (in that they may contain four bytes of data and a byte code and may be queued for delivery), this functionality can be explicitly selected or deselected on a per-signal basis, allowing this converged implementation to still comply with the standard.

Incidentally, the UNIX-style signals can select POSIX realtime signal queuing, if the application wants it. The QNX Neutrino RTOS also extends the signal-delivery mechanisms of POSIX by allowing signals to be targeted at specific threads, rather than simply at the process containing the threads. Since signals are an asynchronous event, they're also implemented with the event-delivery mechanisms.

Microkernel call	POSIX call	Description
SignalKill()	*kill()*, *pthread_kill()*, *raise()*, *sigqueue()*	Set a signal on a process group, process, or thread.
SignalAction()	*sigaction()*	Define action to take on receipt of a signal.
SignalProcmask()	*sigprocmask()*, *pthread_sigmask()*	Change signal blocked mask of a thread.
SignalSuspend()	*sigsuspend()*, *pause()*	Block until a signal invokes a signal handler.
SignalWaitinfo()	*sigwaitinfo()*	Wait for signal and return info on it.

The original POSIX specification defined signal operation on processes only. In a multithreaded process, the following rules are followed:

- Signals caused by CPU exceptions (e.g., SIGSEGV, SIGBUS) are always delivered to the thread that caused the exception.
- The signal actions are maintained at the process level. If a thread ignores or catches a signal, it affects *all* threads within the process.
- The signal mask is maintained at the thread level. If a thread blocks a signal, it affects only that thread.
- An unignored signal targeted at a thread will be delivered to that thread alone.
- An unignored signal targeted at a process is delivered to the first thread that doesn't have the signal blocked. If all threads have the signal blocked, the signal will be queued on the process until any thread ignores or unblocks the signal. If ignored, the signal on the process will be removed. If unblocked, the signal will be moved from the process to the thread that unblocked it.

When a signal is targeted at a process with a large number of threads, the thread table must be scanned, looking for a thread with the signal unblocked. Standard practice for most multithreaded processes is to mask the signal in all threads but one, which is dedicated to handling them. To increase the efficiency of process-signal delivery, the kernel will cache the last thread that accepted a signal and will always attempt to deliver the signal to it first.

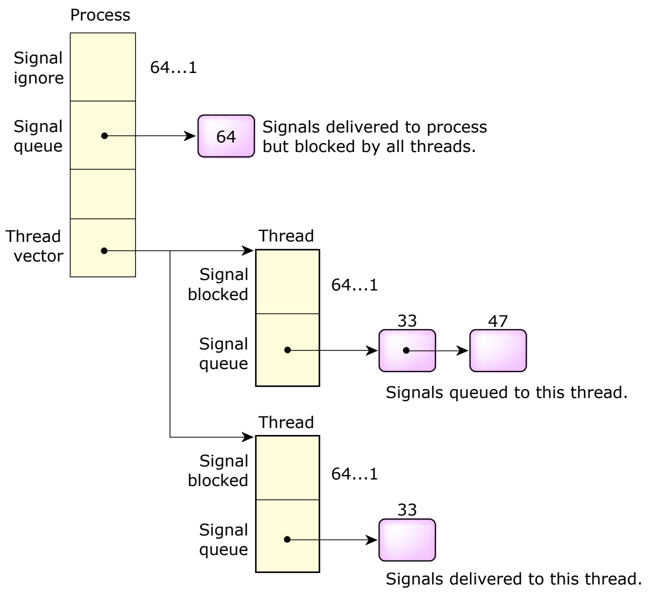


Figure 27: Signal delivery.

The POSIX standard includes the concept of queued realtime signals. The QNX Neutrino RTOS supports optional queuing of any signal, not just realtime signals. The queuing can be specified on a signal-by-signal basis within a process. Each signal can have an associated 8-bit code and a 32-bit value.

This is very similar to message pulses described earlier. The kernel takes advantage of this similarity and uses common code for managing both signals and pulses. The signal number is mapped to a pulse priority using _SIGMAX – *signo*. As a result, signals are delivered in priority order with *lower* signal numbers having *higher* priority. This conforms with the POSIX standard, which states that existing signals have priority over the new realtime signals.

It isn't safe to use floating-point operations in signal handlers.

Special signals

As mentioned earlier, the OS defines a total of 64 signals.

Their range is as follows:

Signal range	Description
1 ... 57	57 POSIX signals (including traditional UNIX signals)
41 ... 56	16 POSIX realtime signals (SIGRTMIN to SIGRTMAX)
57 ... 64	Eight special-purpose QNX Neutrino signals

The eight special signals cannot be ignored or caught. An attempt to call the *signal()* or *sigaction()* functions or the *SignalAction()* kernel call to change them will fail with an error of EINVAL.

In addition, these signals are always blocked and have signal queuing enabled. An attempt to unblock these signals via the *sigprocmask()* function or *SignalProcmask()* kernel call will be quietly ignored.

A regular signal can be programmed to this behavior using the following standard signal calls. The special signals save the programmer from writing this code and protect the signal from accidental changes to this behavior.

```
sigset_t *set;
struct sigaction action;

sigemptyset(&set);
sigaddset(&set, signo);
sigprocmask(SIG_BLOCK, &set, NULL);

action.sa_handler = SIG_DFL;
action.sa_flags = SA_SIGINFO;
sigaction(signo, &action, NULL);
```

This configuration makes these signals suitable for synchronous notification using the *sigwaitinfo()* function or *SignalWaitinfo()* kernel call. The following code will block until the eighth special signal is received:

```
sigset_t *set;
siginfo_t info;

sigemptyset(&set);
sigaddset(&set, SIGRTMAX + 8);
sigwaitinfo(&set, &info);
printf("Received signal %d with code %d and value %d\n",
            info.si_signo,
            info.si_code,
            info.si_value.sival_int);
```

Since the signals are always blocked, the program cannot be interrupted or killed if the special signal is delivered outside of the *sigwaitinfo()* function. Since signal queuing is always enabled, signals won't be lost—they'll be queued for the next *sigwaitinfo()* call.

These signals were designed to solve a common IPC requirement where a server wishes to notify a client that it has information available for the client. The server will use the *MsgDeliverEvent()* call to notify the client. There are two reasonable choices for the event within the notification: pulses or signals.

A pulse is the preferred method for a client that may also be a server to other clients. In this case, the client will have created a channel for receiving messages and can also receive the pulse.

This won't be true for most simple clients. In order to receive a pulse, a simple client would be forced to create a channel for this express purpose. A signal can be used in place of a pulse if the signal is configured to be synchronous (i.e., the signal is blocked) and queued—this is exactly how the special signals are configured. The client would replace the *MsgReceive()* call used to wait for a pulse on a channel with a simple *sigwaitinfo()* call to wait for the signal.

The eight special signals include named signals for special purposes:

SIGSELECT

Used by *select()* to wait for I/O from multiple servers.

Summary of signals

This table describes what each signal means.

Signal	Description	Default action
SIGABRT	Abnormal termination, issued by functions such as *abort()*	Kill the process and write a dump file
SIGALRM	Alarm clock, issued by functions such as *alarm()*	Kill the process
SIGBUS	Bus error, or a memory parity error (a QNX Neutrino-specific interpretation). If a second fault occurs while your process is in a signal handler for this fault, the process is terminated.	Kill the process and write a dump file
SIGCHLD or SIGCLD	A child process terminated	Ignore the signal, but still let the process's children become zombies
SIGCONT	Continue the process. You can't block this signal.	Make the process continue if it's STOPPED; otherwise ignore the signal
SIGDEADLK	A mutex deadlock occurred. If a process dies while holding a mutex, and you haven't called *SyncMutexEvent()* to set up an event to be delivered to the mutex's owner when the mutex dies, the kernel delivers a SIGDEADLK to all threads that are waiting on the mutex without a timeout. SIGDEADLK and SIGEMT refer to the same signal. Some utilities (e.g., `gdb`, `ksh`, `slay`, and `kill`) know about SIGEMT, but not SIGDEADLK.	Kill the process and write a dump file
SIGEMT	EMT instruction (emulation trap) SIGDEADLK and SIGEMT refer to the same signal. Some utilities (e.g., `gdb`, `ksh`, `slay`, and `kill`) know about SIGEMT, but not SIGDEADLK.	Kill the process and write a dump file
SIGFPE	Floating point exception	Kill the process and write a dump file
SIGHUP	Hangup; the session leader died, or the controlling terminal closed	Kill the process

Signal	Description	Default action
SIGILL[a]	Illegal hardware instruction. If a second fault occurs while your thread is in a signal handler for this fault, the process is terminated.	Kill the process and write a dump file
SIGINT	Interrupt; typically generated when you press **Ctrl–C** or **Ctrl–Break** (you can change this with `stty`)	Kill the process
SIGIO	Asynchronous I/O	Ignore the signal
SIGIOT	I/O trap; a synonym for SIGABRT	Kill the process
SIGKILL	Kill. You can't block or catch this signal.	Kill the process
SIGPIPE	Write on pipe with no reader	Kill the process
SIGPOLL	System V name for SIGIO	Ignore the signal
SIGPROF	Profiling timer expired. POSIX has marked this signal as obsolescent; QNX Neutrino doesn't support profiling timers or send this signal.	Kill the process
SIGPWR	Power failure	Ignore the signal
SIGQUIT	Quit; typically generated when you press **Ctrl–\** (you can change this with `stty`)	Kill the process and write a dump file
SIGSEGV	Segmentation violation; an invalid memory reference was detected. If a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.	Kill the process and write a dump file
SIGSTOP	Stop the process. You can't block or catch this signal.	Stop the process
SIGSYS	Bad argument to system call	Kill the process and write a dump file
SIGTERM	Termination signal	Kill the process
SIGTRAP	Trace trap	Kill the process and write a dump file
SIGTSTP	Stop signal from tty; typically generated when you press **Ctrl–Z** (you can change this with `stty`)	Stop the process
SIGTTIN	Background read attempted from control terminal	Stop the process
SIGTTOU	Background write attempted to control terminal	Stop the process
SIGURG	Urgent condition on I/O channel	Ignore the signal
SIGUSR1	User-defined signal 1	Kill the process

Signal	Description	Default action
SIGUSR2	User-defined signal 2	Kill the process
SIGVTALRM	Virtual timer expired. POSIX has marked this signal as obsolescent; QNX Neutrino doesn't support virtual timers or send this signal.	Kill the process
SIGWINCH	The size of the terminal window changed	Ignore the signal
SIGXCPU	Soft CPU time limit exceeded see the RLIMIT_CPU resource for *setrlimit()*)	Kill the process and write a dump file

[a] One possible cause for a SIGILL signal is trying to perform an operation that requires *I/O privileges*. A thread can request these privileges by making sure the process has the PROCMGR_AID_IO ability enabled (see *procmgr_ability()*) and then calling *ThreadCtl()*) specifying the _NTO_TCTL_IO flag:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

POSIX message queues

POSIX defines a set of nonblocking message-passing facilities known as *message queues*.

Like pipes, message queues are named objects that operate with "readers" and "writers." As a priority queue of discrete messages, a message queue has more structure than a pipe and offers applications more control over communications. POSIX message queues provide a familiar interface for many realtime programmers, in that they're similar to the "mailboxes" found in many realtime executives.

In order for you to use POSIX message queues, the message queue server must be running. QNX Neutrino has two implementations of message queues:

- a "traditional" implementation that uses the `mqueue` resource manager (see the "*Resource Managers*" chapter in this book)
- an alternate implementation that uses the `mq` server and queues in kernel space

For more information about these implementations, see the *Utilities Reference* and the "POSIX Message Queues: Two Implementations" technote.

There's a fundamental difference between QNX Neutrino native messages and POSIX message queues: our messages block—they copy their data directly between the address spaces of the processes sending the messages. POSIX message queues, on the other hand, implement a store-and-forward design in which the sender need not block and may have many outstanding messages queued. POSIX message queues exist independently of the processes that use them. You could use message queues in a design where a number of named queues will be operated on by a variety of processes over time.

Which should you use? Message queues, being defined by POSIX, are more portable, but native messages have several advantages:

- better performance
- unbounded, non-fixed message length
- priority inheritance
- known senders

Message queues resemble files, at least as far as their interface is concerned. POSIX defines the following functions that you can use to manage message queues (see the QNX Neutrino *C Library Reference* for more information):

Function	Description
mq_open()	Open a message queue
mq_close()	Close a message queue
mq_unlink()	Remove a message queue
mq_send()	Add a message to the message queue
mq_receive()	Receive a message from the message queue

Function	Description
mq_notify()	Tell the calling process that a message is available on a message queue
mq_setattr()	Set message queue attributes
mq_getattr()	Get message queue attributes

For strict POSIX conformance, you should create message queues that start with a single slash (/) and contain no other slashes. But note that we extend the POSIX standard by supporting pathnames that may contain multiple slashes. This allows, for example, a company to place all its message queues under its company name and distribute a product with increased confidence that a queue name will *not* conflict with that of another company.

In QNX Neutrino, all message queues created appear in the filename space under the directory:

- **/dev/mqueue** if you're using the traditional (`mqueue`) implementation
- **/dev/mq** if you're using the alternate (`mq`) implementation

For example, with the traditional implementation:

mq_open() **name:**	**Pathname of message queue:**
/data	**/dev/mqueue/data**
/acme/data	**/dev/mqueue/acme/data**
/qnx/data	**/dev/mqueue/qnx/data**

You can display all message queues in the system using the `ls` command as follows:

```
ls -Rl /dev/mqueue
```

The size printed is the number of messages waiting.

Shared memory

Shared memory offers the highest bandwidth IPC available.

Once a shared-memory object is created, processes with access to the object can use pointers to directly read and write into it. This means that access to shared memory is in itself *unsynchronized*. If a process is updating an area of shared memory, care must be taken to prevent another process from reading or updating the same area. Even in the simple case of a read, the other process may get information that is in flux and inconsistent.

To solve these problems, shared memory is often used in conjunction with one of the synchronization primitives to make updates atomic between processes. If the granularity of updates is small, then the synchronization primitives themselves will limit the inherently high bandwidth of using shared memory. Shared memory is therefore most efficient when used for updating large amounts of data as a block.

Both semaphores and mutexes are suitable synchronization primitives for use with shared memory. Semaphores were introduced with the POSIX realtime standard for interprocess synchronization. Mutexes were introduced with the POSIX threads standard for thread synchronization. Mutexes may also be used between threads in different processes. POSIX considers this an optional capability; we support it. In general, mutexes are more efficient than semaphores.

Shared memory with message passing

Shared memory and message passing can be combined to provide IPC that offers:

- very high performance (shared memory)
- synchronization (message passing)
- network transparency (message passing)

Using message passing, a client sends a request to a server and blocks. The server receives the messages in priority order from clients, processes them, and replies when it can satisfy a request. At this point, the client is unblocked and continues. The very act of sending messages provides natural synchronization between the client and the server. Rather than copy all the data through the message pass, the message can contain a reference to a shared-memory region, so the server could read or write the data directly. This is best explained with a simple example.

Let's assume a graphics server accepts draw image requests from clients and renders them into a frame buffer on a graphics card. Using message passing alone, the client would send a message containing the image data to the server. This would result in a copy of the image data from the client's address space to the server's address space. The server would then render the image and issue a short reply.

If the client didn't send the image data inline with the message, but instead sent a reference to a shared-memory region that contained the image data, then the server could access the client's data *directly*.

Since the client is blocked on the server as a result of sending it a message, the server knows that the data in shared memory is stable and will not change until the server replies. This combination of message passing and shared memory achieves natural synchronization and very high performance.

This model of operation can also be reversed—the server can generate data and give it to a client. For example, suppose a client sends a message to a server that will read video data directly from a DVD into a shared memory buffer provided by the client. The client will be blocked on the server while the shared memory is being changed. When the server replies and the client continues, the shared memory will be stable for the client to access. This type of design can be pipelined using more than one shared-memory region.

Simple shared memory can't be used between processes on different computers connected via a network. Message passing, on the other hand, is network transparent. A server could use shared memory for local clients and full message passing of the data for remote clients. This allows you to provide a high-performance server that is also network transparent.

In practice, the message-passing primitives are more than fast enough for the majority of IPC needs. The added complexity of a combined approach need only be considered for special applications with very high bandwidth.

Creating a shared-memory object

Multiple threads within a process share the memory of that process. To share memory between processes, you must first create a shared-memory region and then map that region into your process's address space. Shared-memory regions are created and manipulated using the following calls:

Function	Description	Classification
shm_open()	Open (or create) a shared-memory region.	POSIX
close()	Close a shared-memory region.	POSIX
mmap()	Map a shared-memory region into a process's address space.	POSIX
munmap()	Unmap a shared-memory region from a process's address space.	POSIX
munmap_flags()	Unmap previously mapped addresses, exercising more control than possible with *munmap()*	QNX Neutrino
mprotect()	Change protections on a shared-memory region.	POSIX
msync()	Synchronize memory with physical storage.	POSIX
shm_ctl(), *shm_ctl_special()*	Give special attributes to a shared-memory object.	QNX Neutrino
shm_unlink()	Remove a shared-memory region.	POSIX

POSIX shared memory is implemented in the QNX Neutrino RTOS via the process manager (`procnto`). The above calls are implemented as messages to `procnto` (see the *Process Manager* chapter in this book).

The *shm_open()* function takes the same arguments as *open()* and returns a file descriptor to the object. As with a regular file, this function lets you create a new shared-memory object or open an existing shared-memory object.

You must open the file descriptor for reading; if you want to write in the memory object, you also need write access, unless you specify a private (MAP_PRIVATE) mapping.

When a new shared-memory object is created, the size of the object is set to zero. To set the size, you use *ftruncate()*—the very same function used to set the size of a file—or *shm_ctl()*.

mmap()

Once you have a file descriptor to a shared-memory object, you use the *mmap()* function to map the object, or part of it, into your process's address space.

The *mmap()* function is the cornerstone of memory management within QNX Neutrino and deserves a detailed discussion of its capabilities.

You can also use *mmap()* to map files and typed memory objects into your process's address space.

The *mmap()* function is defined as follows:

```
void * mmap( void *where_i_want_it,
             size_t length,
             int memory_protections,
             int mapping_flags,
             int fd,
             off_t offset_within_shared_memory );
```

In simple terms this says: “Map in *length* bytes of shared memory at *offset_within_shared_memory* in the shared-memory object associated with *fd*.”

The *mmap()* function will try to place the memory at the address *where_i_want_it* in your address space. The memory will be given the protections specified by *memory_protections* and the mapping will be done according to the *mapping_flags*.

The three arguments *fd*, *offset_within_shared_memory*, and *length* define a portion of a particular shared object to be mapped in. It's common to map in an entire shared object, in which case the offset will be zero and the length will be the size of the shared object in bytes. On an Intel processor, the length will be a multiple of the page size, which is 4096 bytes.

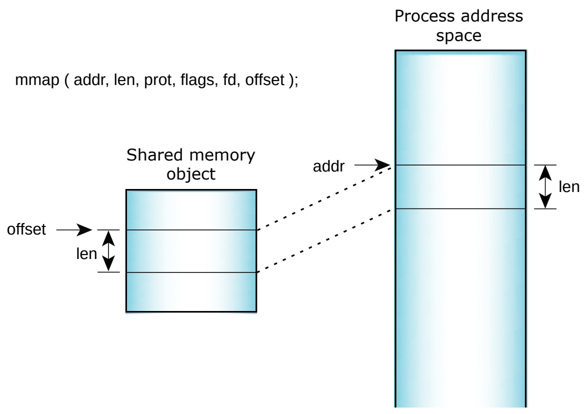


Figure 28: Mapping memory with *mmap()*.

The return value of *mmap()* will be the address in your process's address space where the object was mapped. The argument *where_i_want_it* is used as a hint by the system to where you want the object placed. If possible, the object will be placed at the address requested. Most applications specify an address of zero, which gives the system free rein to place the object where it wishes.

The following protection types may be specified for *memory_protections*:

Manifest	Description
PROT_EXEC	Memory may be executed.
PROT_NOCACHE	Memory should not be cached.
PROT_NONE	No access allowed.
PROT_READ	Memory may be read.
PROT_WRITE	Memory may be written.

You should use the PROT_NOCACHE manifest when you're using a shared-memory region to gain access to dual-ported memory that may be modified by hardware (e.g., a video frame buffer or a memory-mapped network or communications board). Without this manifest, the processor may return "stale" data from a previously cached read.

The *mapping_flags* determine how the memory is mapped. These flags are broken down into two parts—the first part is a type and must be specified as one of the following:

Map type	Description
MAP_SHARED	The mapping may be shared by many processes; changes are propagated back to the underlying object.
MAP_PRIVATE	The mapping is private to the calling process; changes *aren't* propagated back to the underlying object. The *mmap()* function allocates system RAM and makes a copy of the object.

The MAP_SHARED type is the one to use for setting up shared memory between processes; MAP_PRIVATE has more specialized uses.

Don't have a shared, writable mapping to a file that you're simultaneously accessing via *write()*. The interaction between the two methods isn't well defined and may give unexpected results.

You can OR a number of flags into the above type to further define the mapping. These are described in detail in the *mmap()* entry in the QNX Neutrino *C Library Reference*. A few of the more interesting flags are:

MAP_ANON

Map anonymous memory that isn't associated with any file descriptor; you must set the *fd* parameter to NOFD. The *mmap()* function allocates the memory and fills it with zeros.

You commonly use MAP_ANON with MAP_SHARED to create a shared memory area for forked applications. You can use MAP_ANON as the basis for a page-level memory allocator.

MAP_FIXED

Map the object to the address specified by *where_i_want_it*. If a shared-memory region contains pointers within it, then you may need to force the region at the same address in all processes that map it. This can be avoided by using offsets within the region in place of direct pointers.

MAP_PHYS

This flag indicates that you wish to deal with physical memory. The *fd* parameter should be set to NOFD. When used without MAP_ANON, the *offset_within_shared_memory* specifies the exact physical address to map (e.g., for video frame buffers).

If used with MAP_ANON, then physically contiguous memory is allocated (e.g., for a DMA buffer). You typically use this combination with MAP_SHARED.

You should use *mmap_device_memory()* instead of MAP_PHYS, unless you're allocating physically contiguous memory.

Using the mapping flags described above, a process can easily share memory between processes:

```
/* Map in a shared memory region */
fd = shm_open("datapoints", O_RDWR);
addr = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

or allocate a DMA buffer for a bus-mastering PCI network card:

```
/* Allocate a physically contiguous buffer */
addr = mmap(0, 262144, PROT_READ | PROT_WRITE | PROT_NOCACHE,
            MAP_SHARED | MAP_PHYS | MAP_ANON, NOFD, 0);
```

You can unmap all or part of a shared-memory object from your address space using *munmap()*. This primitive isn't restricted to unmapping shared memory—it can be used to unmap any region of memory within your process. When used in conjunction with the MAP_ANON flag to *mmap()*, you can easily implement a private page-level allocator/deallocator.

You can change the protections on a mapped region of memory using *mprotect()*. Like *munmap()*, *mprotect()* isn't restricted to shared-memory regions—it can change the protection on any region of memory within your process.

Typed memory

Typed memory is POSIX functionality defined in the 1003.1 specification. It's part of the advanced realtime extensions, and the manifests are located in the **<sys/mman.h>** header file.

Typed memory adds the following functions to the C library:

posix_typed_mem_open()

Open a typed memory object. This function returns a file descriptor, which you can then pass to *mmap()* to establish a memory mapping of the typed memory object.

posix_typed_mem_get_info()

Get information (currently the amount of available memory) about a typed memory object.

POSIX typed memory provides an interface to open memory objects (which are defined in an OS-specific fashion) and perform mapping operations on them. It's useful in providing an abstraction between BSP- or board-specific address layouts and device drivers or user code.

Implementation-defined behavior

POSIX specifies that typed memory pools (or objects) are created and defined in an implementation-specific fashion.

This section describes the following for QNX Neutrino:

- *Seeding of typed memory regions*
- *Naming of typed memory regions*
- *Pathname space and typed memory*
- *mmap() allocation flags and typed memory objects*
- *Permissions and typed memory objects*
- *Object length and offset definitions*
- *Interaction with other POSIX APIs*

Seeding of typed memory regions

Under QNX Neutrino, typed memory objects are defined from the memory regions specified in the *asinfo* section of the system page. Thus, typed memory objects map directly to the address space hierarchy (*asinfo* segments) define by startup. The typed memory objects also inherit the properties defined in *asinfo*, namely the physical address (or bounds) of the memory segments.

In general, the naming and properties of the *asinfo* entries is arbitrary and is completely under the user's control. There are, however, some mandatory entries:

`memory`

Physical addressability of the processor, typically 4 GB on a 32-bit CPU (more with physical addressing extensions).

`ram`

All of the RAM on the system. This may consist of multiple entries.

`sysram`

System RAM, i.e., memory that has been given to the OS to manage. This may also consist of multiple entries.

Since by convention `sysram` is the memory that has been given to the OS, this pool is the same as that used by the OS to satisfy anonymous *mmap()* and *malloc()* requests.

You can create additional entries, but only in startup, using the *as_add()* function.

Naming of typed memory regions

The names of typed memory regions are derived directly from the names of the *asinfo* segments. The *asinfo* section itself describes a hierarchy, and so the naming of typed memory object is a hierarchy.

Here's a sample system configuration:

Name	Range (start, end)
`/memory`	0, 0xFFFFFFFF
`/memory/ram`	0, 0x1FFFFFF
`/memory/ram/sysram`	0x1000, 0x1FFFFFF
`/memory/isa/ram/dma`	0x1000, 0xFFFFFF
`/memory/ram/dma`	0x1000, 0x1FFFFFF

The name you pass to *posix_typed_mem_open()* follows the above naming convention. POSIX allows an implementation to define what happens when the name doesn't start with a leading slash (`/`). The resolution rules on opening are as follows:

1. If the name starts with a leading `/`, an exact match is done.
2. The name may contain intermediate `/` characters. These are considered as path component separators. If multiple path components are specified, they're matched from the bottom up (the opposite of the way filenames are resolved).
3. If the name doesn't start with a leading `/`, a tail match is done on the pathname components specified.

Here are some examples of how *posix_typed_mem_open()* resolves names, using the above sample configuration:

This name:	Resolves to:	See:
`/memory`	`/memory`	Rule 1
`/memory/ram`	`/memory/ram`	Rule 2

This name:	Resolves to:	See:
`/sysram`	Fails	
`sysram`	`/memory/ram/sysram`	Rule 3

Pathname space and typed memory

The typed memory name hierarchy is exported through the process manager namespace under **/dev/tymem**. Applications can list this hierarchy, and look at the *asinfo* entries in the system page to get information about the typed memory.

Unlike for shared memory objects, you can't open typed memory through the namespace interface, because *posix_typed_mem_open()* takes the additional parameter *tflag*, which is required and isn't provided in the *open()* API.

mmap() allocation flags and typed memory objects

The following general cases of allocations and mapping are considered for typed memory:

- The typed memory pool is explicitly allocated from (POSIX_TYPED_MEM_ALLOCATE and POSIX_TYPED_MEM_ALLOCATE_CONTIG). This case is just like a normal MAP_SHARED of an anonymous object:

  ```
  mmap(0, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON,
       NOFD, 0);
  ```

  The memory is allocated and not available for other allocations, but if you fork the process, the child processes can access it as well. The memory is released when the last mapping to it is removed.

  Note that like somebody doing *mem_offset()* and then a MAP_PHYS to gain access to previously allocated memory, somebody else could open the typed memory object with POSIX_TYPED_MEM_MAP_ALLOCATABLE (or with no flags) and gain access to the same physical memory that way.

  POSIX_TYPED_MEM_ALLOCATE_CONTIG is like `MAP_ANON | MAP_SHARED`, in that it causes a contiguous allocation.

- The POSIX_TYPED_MEM_MAP_ALLOCATABLE case, which is used to create a mapping to an object without allocation or deallocation. This is equivalent to a shared mapping to physical memory.

You should use only MAP_SHARED mappings, since a write to a MAP_PRIVATE mapping will (as normal) create a private copy for the process in normal anonymous memory.

If you specify no flag, or you specify POSIX_TYPED_MEM_MAP_ALLOCATABLE, the offset parameter to *mmap()* specifies the starting physical address in the typed memory region; if the typed memory region is discontiguous (multiple *asinfo* entries), the allowed offset values are also discontiguous and don't start at zero as they do for shared memory objects. If you specify a [*paddr*, *paddr* + *size*) region that falls outside the allowed addresses for the typed memory object, *mmap()* fails with ENXIO.

Permissions and typed memory objects

Permissions on a typed memory object are governed by UNIX permissions.

The *oflags* argument to *posix_typed_mem_open()* specifies the desired access privilege, and these flags are checked against the permission mask of the typed memory object.

POSIX doesn't specify how permissions are assigned to the typed memory objects. Under QNX Neutrino, default permissions are assigned at system boot-up. By default, **root** is the owner and group, and has read-write permissions; no one else has any permissions.

Currently, there's no mechanism to change the permissions of an object. In the future, the implementation may be extended to allow *chmod()* and *chown()* to modify the permissions.

Object length and offset definitions

You can retrieve the size of an object by using *posix_typed_mem_get_info()*.

The *posix_typed_mem_get_info()* call fills in a `posix_typed_mem_info` structure, which includes the *posix_tmi_length* field, which contains the size of the typed memory object.

As specified by POSIX, the length field is dynamic and contains the current allocatable size for that object (in effect, the free size of the object for POSIX_TYPED_MEM_ALLOCATE and POSIX_TYPED_MEM_ALLOCATE_CONTIG). If you opened the object with a *tflag* of 0 or POSIX_TYPED_MEM_MAP_ALLOCATABLE, the length field is set to zero.

When you map in a typed memory object, you usually pass an offset to *mmap()*. The offset is the physical address of the location in the object where the mapping should commence. The offset is appropriate only when opening the object with a *tflag* of 0 or POSIX_TYPED_MEM_MAP_ALLOCATABLE. If you opened the typed memory object with POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG, a nonzero offset causes the call to *mmap()* to fail with an error of EINVAL.

Interaction with other POSIX APIs

Typed memory can interact with other POSIX APIs.

`rlimits`

The POSIX *setrlimit()* APIs provide the ability to set limits on the virtual and physical memory that a process can consume. Since typed memory operations may operate on normal RAM (`sysram`) and will create mappings in the process's address space, they need to be taken into account when doing the `rlimit` accounting. In particular, the following rules apply:

- Any mapping created by *mmap()* for typed memory objects is counted in the process's RLIMIT_VMEM or RLIMIT_AS limit.
- Typed memory never counts against RLIMIT_DATA.

POSIX file-descriptor functions

You can use the file descriptor that *posix_typed_memory_open()* returns with selected POSIX fd-based calls, as follows:

- *fstat(fd,..)*, which fills in the `stat` structure as it does for a shared memory object. In QNX Neutrino 7.0 or later, the *st_size* field holds the size of the typed memory object. In earlier releases, it was used internally by the OS.
- *close(fd)* closes the file descriptor.
- *dup()* and *dup2()* duplicate the file handle.
- *posix_mem_offset()* behaves as documented in the POSIX specification.

Practical examples

Here are some examples of how you could use typed memory.

Allocating contiguous memory from system RAM

Here's a code snippet that allocates contiguous memory from system RAM:

```
int fd = posix_typed_mem_open( "/memory/ram/sysram", O_RDWR,
            POSIX_TYPED_MEM_ALLOCATE_CONTIG);

void *vaddr = mmap( NULL, size, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE, fd, 0);
```

Defining packet memory and allocating from it

Assume you have special memory (say fast SRAM) that you want to use for packet memory. This SRAM isn't put in the global system RAM pool. Instead, in startup, we use *as_add()* (see the Customizing Image Startup Programs chapter of *Building Embedded Systems*) to add an *asinfo* entry for the packet memory:

```
as_add(phys_addr, phys_addr + size - 1, AS_ATTR_NONE,
       "packet_memory", mem_id);
```

where *phys_addr* is the physical address of the SRAM, *size* is the SRAM size, and *mem_id* is the ID of the parent (typically `memory`, which is returned by *as_default()*).

This code creates an *asinfo* entry for `packet_memory`, which you can then use as POSIX typed memory. The following code allows different applications to allocate pages from `packet_memory`:

```
int fd = posix_typed_mem_open( "packet_memory", O_RDWR,
            POSIX_TYPED_MEM_ALLOCATE);
void *vaddr = mmap( NULL, size, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
```

Alternatively, you may want to use the packet memory as direct shared, physical buffers. In this case, applications would use it as follows:

```
int fd = posix_typed_mem_open( "packet_memory", O_RDWR,
            POSIX_TYPED_MEM_MAP_ALLOCATABLE);
void *vaddr = mmap( NULL, size, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, offset);
```

Defining a DMA-safe region

On some hardware, due to limitations of the chipset or memory controller, it may not be possible to perform DMA to arbitrary addresses in the system. In some cases, the chipset has only the ability to DMA to a subset of all physical RAM. This has traditionally been difficult to solve without statically reserving some portion of RAM of driver DMA buffers (which is potentially wasteful). Typed memory provides a clean abstraction to solve this issue. Here's an example:

In startup, use *as_add_containing()* (see the Customizing Image Startup Programs chapter of *Building Embedded Systems*) to define an *asinfo* entry for DMA-safe memory. Make this entry be a child of `ram`:

```
as_add_containing( dma_addr, dma_addr + size - 1,
                   AS_ATTR_RAM, "dma", "ram");
```

where *dma_addr* is the start of the DMA-safe RAM, and *size* is the size of the DMA-safe region.

This code creates an *asinfo* entry for `dma`, which is a child of `ram`. Drivers can then use it to allocate DMA-safe buffers:

```
int fd = posix_typed_mem_open( "ram/dma", O_RDWR,
            POSIX_TYPED_MEM_ALLOCATE_CONTIG);
void *vaddr = mmap( NULL, size, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
```

Pipes and FIFOs

Pipes and FIFOs are both forms of queues that connect processes.

In order for you to use pipes or FIFOs in the QNX Neutrino RTOS, the pipe resource manager (`pipe`) must be running.

Pipes

A *pipe* is an unnamed file that serves as an I/O channel between two or more cooperating processes: one process writes into the pipe, the other reads from the pipe.

The `pipe` manager takes care of buffering the data. The buffer size is defined as PIPE_BUF in the **<limits.h>** file. A pipe is removed once both of its ends have closed. The function *pathconf()* returns the value of the limit.

Pipes are normally used when two processes want to run in parallel, with data moving from one process to the other in a single direction. (If bidirectional communication is required, messages should be used instead.)

A typical application for a pipe is connecting the output of one program to the input of another program. This connection is often made by the shell. For example:

```
ls | more
```

directs the standard output from the `ls` utility through a pipe to the standard input of the `more` utility.

If you want to:	Use the:
Create pipes from within the shell	pipe symbol ("|")
Create pipes from within programs	*pipe()* or *popen()* functions

FIFOs

FIFOs are essentially the same as pipes, except that FIFOs are named permanent files that are stored in filesystem directories.

If you want to:	Use the:
Create FIFOs from within the shell	`mkfifo` utility
Create FIFOs from within programs	*mkfifo()* function
Remove FIFOs from within the shell	`rm` utility
Remove FIFOs from within programs	*remove()* or *unlink()* function

Chapter 4
The Instrumented Microkernel

An instrumented version of the microkernel (`procnto-instr`) is equipped with a sophisticated tracing and profiling mechanism that lets you monitor your system's execution in real time. The `procnto-instr` module works on both single-CPU and SMP systems.

The `procnto-instr` module uses very little overhead and gives exceptionally good performance—it's typically about 98% as fast as the noninstrumented kernel (when it isn't logging). The additional amount of code (about 30 KB on an x86 system) in the instrumented kernel is a relatively small price to pay for the added power and flexibility of this useful tool. Depending on the footprint requirements of your final system, you may choose to use this special kernel as a development/prototyping tool or as the actual kernel in your final product.

The instrumented module is nonintrusive—you don't have to modify a program's source code in order to monitor how that program interacts with the kernel. You can trace as many or as few interactions (e.g., kernel calls, state changes, and other system activities) as you want between the kernel and any running thread or process in your system. You can even monitor interrupts. In this context, all such activities are known as *events*.

For more details, see the System Analysis Toolkit *User's Guide*.

Instrumentation at a glance

Here are the essential tasks involved in kernel instrumentation:

1. The instrumented microkernel (`procnto-instr`) emits trace events as a result of various system activities. These events are automatically copied to a set of buffers grouped into a circular linked list.
2. As soon as the number of events inside a buffer reaches the high-water mark, the kernel notifies a data-capture utility.
3. The data-capture utility then writes the trace events from the buffer to an output device (e.g., a serial port, an event file, etc.).
4. A data-interpretation facility then interprets the events and presents this data to the user.

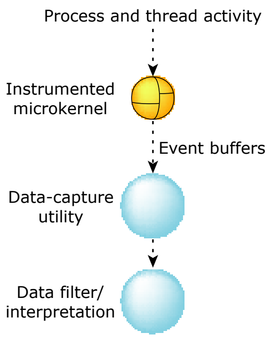


Figure 29: Instrumentation at a glance.

Event control

Given the large number of activities occurring in a live system, the number of events that the kernel emits can be overwhelming (in terms of the amount of data, the processing requirements, and the resources needed to store it). But you can easily control the amount of data emitted.

Specifically, you can:

- control the initial conditions that trigger event emissions
- apply predefined kernel filters to dynamically control emissions
- implement your own event handlers for even more filtering.

Once the data has been collected by the data-capture utility (`tracelogger`), it can then be analyzed. You can analyze the data in real time or offline after the relevant events have been gathered. The System Analysis tool within the IDE presents this data graphically so you can "see" what's going on in your system.

Modes of emission

Apart from applying the various filters to control the event stream, you can also specify one of two modes the kernel can use to emit events:

***fast* mode**

Emits only the most pertinent information (e.g., only two kernel call arguments) about an event.

***wide* mode**

Generates more information (e.g., *all* kernel call arguments) for the same event.

The trade-off here is one of speed vs knowledge: fast mode delivers less data, while wide mode packs much more information for each event. Either way, you can easily tune your system, because these modes work on a per-event basis.

As an example of the difference between the fast and wide emission modes, let's look at the kinds of information we might see for a *MsgSendv()* call entry:

Fast mode data	Number of bytes for the event
Connection ID	4 bytes
Message data	4 bytes (the first 4 bytes usually comprise the header)
	Total emitted: 8 bytes

Wide mode data	Number of bytes for the event
Connection ID	4 bytes

Wide mode data	Number of bytes for the event
# of parts to send	4 bytes
# of parts to receive	4 bytes
Message data	4 bytes (the first 4 bytes usually comprise the header)
Message data	4 bytes
Message data	4 bytes
	Total emitted: 24 bytes

Ring buffer

Rather than always emit events to an external device, the kernel can keep all of the trace events in an *internal circular buffer.*

This buffer can be programmatically dumped to an external device on demand when a certain triggering condition is met, making this a very powerful tool for identifying elusive bugs that crop up under certain runtime conditions.

Data interpretation

The data of an event includes a high-precision timestamp as well as the ID number of the CPU on which the event was generated. This information helps you easily diagnose difficult timing problems, which are more likely to occur on multiprocessor systems.

The event format also includes the CPU platform (e.g., x86, ARM, etc.) and endian type, which facilitates remote analysis (whether in real time or offline). Using a data interpreter, you can view the data output in various ways, such as:

- a timestamp-based linear presentation of the entire system
- a “running” view of only the active threads/processes
- a state-based view of events per process/thread.

The linear output from the data interpreter might look something like this:

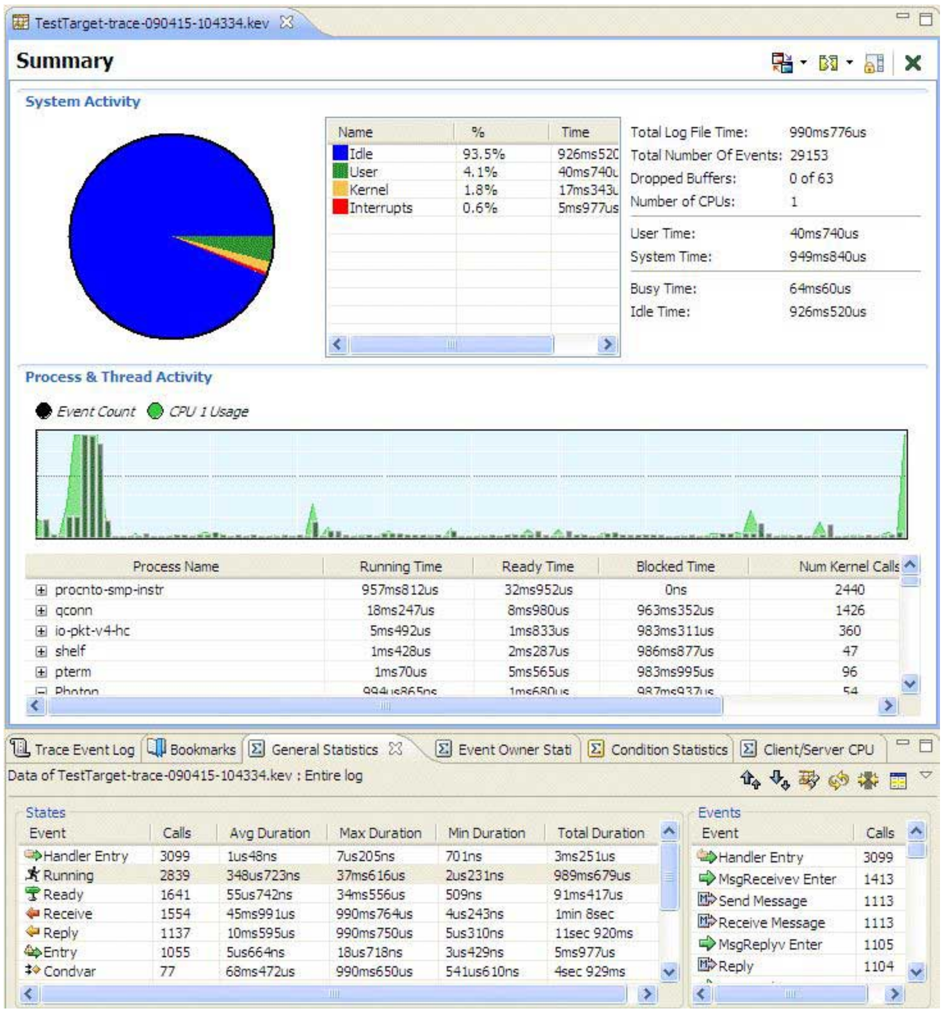
```
TRACEPRINTER version 0.94
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: /dev/shmem/tracebuffer
            TRACE_DATE:: Fri Jun  8 13:14:40 2001
       TRACE_VER_MAJOR:: 0
       TRACE_VER_MINOR:: 96
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Fri Jun  8 04:31:05 2001
  TRACE_CYCLES_PER_SEC:: 400181900
         TRACE_CPU_NUM:: 4
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: x86quad.gp.qa
     TRACE_SYS_RELEASE:: 6.1.0
     TRACE_SYS_VERSION:: 2001/06/04-14:07:56
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2440
 -- KERNEL EVENTS --
t:0x1310da15 CPU:01 CONTROL :TIME msb:0x0000000f, lsb(offset):0x1310d81c
t:0x1310e89d CPU:01 PROCESS :PROCCREATE_NAME
                      ppid:0
                       pid:1
                      name:./procnto-smp-instr
t:0x1310eee4 CPU:00 THREAD  :THCREATE      pid:1 tid:1
t:0x1310f052 CPU:00 THREAD  :THRUNNING     pid:1 tid:1
t:0x1310f144 CPU:01 THREAD  :THCREATE      pid:1 tid:2
t:0x1310f201 CPU:01 THREAD  :THREADY       pid:1 tid:2
t:0x1310f32f CPU:02 THREAD  :THCREATE      pid:1 tid:3
t:0x1310f3ec CPU:02 THREAD  :THREADY       pid:1 tid:3
t:0x1310f52d CPU:03 THREAD  :THCREATE      pid:1 tid:4
t:0x1310f5ea CPU:03 THREAD  :THRUNNING     pid:1 tid:4
t:0x1310f731 CPU:02 THREAD  :THCREATE      pid:1 tid:5
.
.
.
```

To help you fine-tune your interpretation of the event data stream, we provide a library (`traceparser`) so you can write your own custom event interpreters.

System analysis with the IDE

The IDE module of the System Analysis Toolkit (SAT) can serve as a comprehensive instrumentation control and post-processing visualization tool.

From within the IDE, developers can configure all trace events and modes, and then transfer log files automatically to a remote system for analysis. As a visualization tool, the IDE provides a rich set of event and process filters designed to help developers quickly prune down massive event sets in order to see only those events of interest.



Name	%	Time
🟦 Idle	93.5%	926ms520
🟩 User	4.1%	40ms740u
🟧 Kernel	1.8%	17ms343u
🟥 Interrupts	0.6%	5ms977us

Total Log File Time:	990ms776us
Total Number Of Events:	29153
Dropped Buffers:	0 of 63
Number of CPUs:	1
User Time:	40ms740us
System Time:	949ms840us
Busy Time:	64ms60us
Idle Time:	926ms520us

Process Name	Running Time	Ready Time	Blocked Time	Num Kernel Calls
⊞ procnto-smp-instr	957ms812us	32ms952us	0ns	2440
⊞ qconn	18ms247us	8ms980us	963ms352us	1426
⊞ io-pkt-v4-hc	5ms492us	1ms833us	983ms311us	360
⊞ shelf	1ms428us	2ms287us	986ms877us	47
⊞ pterm	1ms70us	5ms565us	983ms995us	96
⊟ Photon	994us865ns	1ms680us	987ms937us	54

Data of TestTarget-trace-090415-104334.kev : Entire log

Event	Calls	Avg Duration	Max Duration	Min Duration	Total Duration
Handler Entry	3099	1us48ns	7us205ns	701ns	3ms251us
Running	2839	348us723ns	37ms616us	2us231ns	989ms679us
Ready	1641	55us742ns	34ms556us	509ns	91ms417us
Receive	1554	45ms991us	990ms764us	4us243ns	1min 8sec
Reply	1137	10ms595us	990ms750us	5us310ns	11sec 920ms
Entry	1055	5us664ns	18us718ns	3us429ns	5ms977us
Condvar	77	68ms472us	990ms650us	541us610ns	4sec 929ms

Event	Calls
Handler Entry	3099
MsgReceivev Enter	1413
Send Message	1113
Receive Message	1113
MsgReplyv Enter	1105
Reply	1104

Figure 30: The IDE helps you visualize system activity.

For more information, see the IDE *User's Guide*.

Proactive tracing

While the instrumented kernel provides an excellent unobtrusive method for instrumenting and monitoring processes, threads, and the state of your system in general, you can also have your applications proactively influence the event-collection process.

Using the *TraceEvent()* library call, applications themselves can inject custom events into the trace stream. This facility is especially useful when building large, tightly coupled, multicomponent systems.

For example, the following simple call would inject the integer values of *eventcode*, *first*, and *second* into the event stream:

```
TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, eventcode, first,
           second);
```

You can also inject a string (e.g., "My Event") into the event stream, as shown in the following code:

```
#include <stdio.h>
#include <sys/trace.h>

/* Code to associate with emitted events */
#define MYEVENTCODE 12

int main(int argc, char **argv) {
    printf("My pid is %d \n", getpid());

    /* Inject two integer events (26, 1975) */
    TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, MYEVENTCODE,
               26, 1975);

    /* Inject a string event (My Event) */
    TraceEvent(_NTO_TRACE_INSERTUSRSTREVENT, MYEVENTCODE,
               "My Event");

    return 0;
}
```

The output, as gathered by the `traceprinter` data interpreter, would then look something like this:

```
.
.
.
t:0x38ea737e CPU:00 USREVENT:EVENT:12, d0:26 d1:1975
.
.
.
t:0x38ea7cb0 CPU:00 USREVENT:EVENT:12 STR:"My Event"
```

Note that 12 was specified as the trace user eventcode for these events.

Chapter 5
Multicore Processing

The QNX Neutrino RTOS can run on single-core or multicore systems. Multiprocessing systems can be in these forms:

Discrete or traditional

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

Multicore

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

Multicore processors deliver greater computing power through concurrency, offer greater system density, and run at lower clock speeds than uniprocessor chips. Multicore processors also reduce thermal dissipation, power consumption, and board area (and hence the cost of the system).

Multiprocessing includes several operating modes:

- *asymmetric*, where a separate OS, or a separate instantiation of the same OS, runs on each CPU.
- *symmetric*, where a single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.
- *bound*, where a single instantiation of an OS manages all CPUs simultaneously, but each application is locked to a specific CPU.

To determine how many processors there are on your system, look at the *num_cpu* entry of the system page. For more information, see the System Page chapter of *Building Embedded Systems*.

Asymmetric multiprocessing (AMP)

Asymmetric multiprocessing provides an execution environment that's similar to conventional uniprocessor systems. It offers a relatively straightforward path for porting legacy code and provides a direct mechanism for controlling how the CPUs are used. In most cases, it lets you work with standard debugging tools and techniques.

AMP can be:

- *homogeneous*—each CPU runs the same type and version of the OS
- *heterogeneous*—each CPU runs either a different OS or a different version of the same OS

QNX Neutrino's distributed programming model lets you make the best use of the multiple CPUs in a homogeneous environment. Applications running on one CPU can communicate transparently with applications and system services (e.g., device drivers, protocol stacks) on other CPUs, without the high CPU utilization imposed by traditional forms of interprocessor communication.

In heterogeneous systems, you must either implement a proprietary communications scheme or choose two OSs that share a common infrastructure (likely IP based) for interprocessor communications. To help avoid resource conflicts, the OSs should also provide standardized mechanisms for accessing shared hardware components.

With AMP, you decide how the shared hardware resources used by applications are divided up between the CPUs. Normally, this resource allocation occurs statically during boot time and includes physical memory allocation, peripheral usage, and interrupt handling. While the system could allocate the resources dynamically, doing so would entail complex coordination between the CPUs.

In an AMP system, a process always runs on the same CPU, even when other CPUs run idle. As a result, one CPU can end up being under- or overutilized. To address the problem, the system could allow applications to migrate dynamically from CPU to another. Doing so, however, can involve complex checkpointing of state information or a possible service interruption as the application is stopped on one CPU and restarted on another. Also, such migration is difficult, if not impossible, if the CPUs run different OSs.

Symmetric multiprocessing (SMP)

Allocating resources in a multicore design can be difficult, especially when multiple software components are unaware of how other components are employing those resources.

Symmetric multiprocessing addresses the issue by running only one copy of the QNX Neutrino RTOS on all of the system's CPUs. Because the OS has insight into all system elements at all times, it can allocate resources on the multiple CPUs with little or no input from the application designer. Moreover, QNX Neutrino provides built-in standardized primitives, such as *pthread_mutex_lock()*, *pthread_mutex_unlock()*, *pthread_spin_lock()*, and *pthread_spin_unlock()*, that let multiple applications share these resources safely and easily.

By running only one copy of QNX Neutrino, SMP can dynamically allocate resources to specific applications rather than to CPUs, thereby enabling greater utilization of available processing power. It also lets system tracing tools gather operating statistics and application interactions for the multiprocessing system as a whole, giving you valuable insight into how to optimize and debug applications.

For instance, the System Profiler in the IDE can track thread migration from one CPU to another, as well as OS primitive usage, scheduling events, application-to-application messaging, and other events, all with high-resolution timestamping. Application synchronization also becomes much easier since you use standard OS primitives rather than complex IPC mechanisms.

QNX Neutrino lets the threads of execution within an application run concurrently on any CPU, making the entire computing power of the chip available to applications at all times. QNX Neutrino's preemption and thread prioritization capabilities help you ensure that CPU cycles go to the application that needs them the most.

In QNX Neutrino 7.0 or later, we require that the hardware underlying *ClockCycles()* be synchronized across all processors on an SMP system. If it isn't, you might encounter some unexpected behavior, such as drifting times and timers.

Booting an x86 SMP system

The microkernel itself contains very little hardware- or system-specific code. The code that determines the capabilities of the system is isolated in a startup program, which is responsible for initializing the system, determining available memory, etc. Information gathered is placed into a memory table available to the microkernel and to all processes (on a read-only basis).

The `startup-x86` program is designed to work on systems that support the Unified Extensible Firmware Interface. This startup program is responsible for:

- determining the number of processors
- determining the address of the local and I/O APIC
- initializing each additional processor

After a reset, only one processor will be executing the reset code. This processor is called the *boot processor* (BP). For each additional processor found, the BP running the `startup-x86` code will:

- initialize the processor
- switch it to 32-bit protected mode
- allocate the processor its own page directory
- set the processor spinning with interrupts disabled, waiting to be released by the kernel

How the SMP microkernel works

Once the additional processors have been released and are running, all processors are considered peers for the scheduling of threads.

Scheduling

The scheduling policy follows the same rules as on a uniprocessor system. That is, the highest-priority thread will be running on an available processor. If a new thread becomes ready to run as the highest-priority thread in the system, it will be dispatched to the appropriate processor. If more than one processor is selected as a potential target, then the microkernel will try to dispatch the thread to the processor where it last ran. This affinity is used as an attempt to reduce thread migration from one processor to another, which can affect cache performance.

In an SMP system, the scheduler has some flexibility in deciding exactly how to schedule the other threads, with an eye towards optimizing cache usage and minimizing thread migration. This could mean that some processors will be running lower-priority threads while a higher-priority thread is waiting to run on the processor it last ran on. The next time a processor that's running a lower-priority thread makes a scheduling decision, it will choose the higher-priority one. In QNX Neutrino 7.0.1 or later, you can use *SchedCtl()* with the SCHED_CONFIGURE command to tune this behavior.

In any case, the realtime scheduling rules that were in place on a uniprocessor system are guaranteed to be upheld on an SMP system.

Kernel locking

In a uniprocessor system, only one thread is allowed to execute within the microkernel at a time. Most kernel operations are short in duration (typically a few microseconds on a Pentium-class processor). The microkernel is also designed to be completely preemptible and restartable for those operations that take more time. This design keeps the microkernel lean and fast without the need for large numbers of fine-grained locks. It is interesting to note that placing many locks in the main code path through a kernel will noticeably slow the kernel down. Each lock typically involves processor bus transactions, which can cause processor stalls.

In an SMP system, QNX Neutrino maintains this philosophy of only one thread in a preemptible and restartable kernel. The microkernel may be entered on any processor, but only one processor will be granted access at a time.

For most systems, the time spent in the microkernel represents only a small fraction of the processor's workload. Therefore, while conflicts will occur, they should be more the exception than the norm. This is especially true for a microkernel where traditional OS services like filesystems are separate processes and not part of the kernel itself.

Interprocessor interrupts (IPIs)

The processors communicate with each other through IPIs (interprocessor interrupts). IPIs can effectively schedule and control threads over multiple processors. For example, an IPI to another processor is often needed when:

- a higher-priority thread becomes ready
- a thread running on another processor is hit with a signal
- a thread running on another processor is canceled
- a thread running on another processor is destroyed

Critical sections

To control access to data structures that are shared between them, threads and processes use the standard POSIX primitives of mutexes, condvars, and semaphores. These work without change in an SMP system.

Many realtime systems also need to protect access to shared data structures between an interrupt handler and the thread that owns the handler. The traditional POSIX primitives used between threads aren't available for use by an interrupt handler. There are two solutions here:

- One is to remove all work from the interrupt handler and do all the work at thread time instead. Given our fast thread scheduling, this is a very viable solution.
- In a uniprocessor system running the QNX Neutrino RTOS, an interrupt handler may preempt a thread, but a thread will never preempt an interrupt handler. This allows the thread to protect itself from the interrupt handler by disabling and enabling interrupts for *very brief* periods of time.

The thread on a non-SMP system protects itself with code of the form:

```
InterruptDisable()
// critical section
InterruptEnable()
```

Or:

```
InterruptMask(intr)
// critical section
InterruptUnmask(intr)
```

Unfortunately, this code will fail on an SMP system since the thread may be running on one processor while the interrupt handler is concurrently running on another processor!

One solution would be to lock the thread to a particular processor (see "*Bound Multiprocessing (BMP)*," later in this chapter).

A better solution would be to use a new exclusion lock available to both the thread and the interrupt handler. This is provided by the following primitives, which work on both uniprocessor and SMP machines:

`InterruptLock(intrspin_t*` ***spinlock*** **`)`**

Attempt to acquire a *spinlock*, a variable shared between the interrupt handler and thread. The code will spin in a tight loop until the lock is acquired. After disabling interrupts, the code will acquire the lock (if it was acquired by a thread). The lock *must* be released as soon as possible (typically within a few lines of C code without any loops).

`InterruptUnlock(intrspin_t*` ***spinlock*** **`)`**

Release a lock and reenable interrupts.

On a non-SMP system, there's no need for a spinlock.

For more information, see the Multicore Processing chapter of the QNX Neutrino *Programmer's Guide*.

Bound multiprocessing (BMP)

Bound multiprocessing provides the scheduling control of an asymmetric multiprocessing model, while preserving the hardware abstraction and management of symmetric multiprocessing.

BMP is similar to SMP, but you can specify which processors a thread can run on. You can use both SMP and BMP on the same system, allowing some threads to migrate from one processor to another, while other threads are restricted to one or more processors.

As with SMP, a single copy of the OS maintains an overall view of all system resources, allowing them to be dynamically allocated and shared among applications. But, during application initialization, a setting determined by the system designer forces all of an application's threads to execute only on a specified CPU.

Compared to full, floating SMP operation, this approach offers several advantages:

- It eliminates the cache thrashing that can reduce performance in an SMP system by allowing applications that share the same data set to run exclusively on the same CPU.
- It offers simpler application debugging than SMP since all execution threads within an application run on a single CPU.
- It helps legacy applications that use poor techniques for synchronizing shared data to run correctly, again by letting them run on a single CPU.

With BMP, an application locked to one CPU can't use other CPUs, even if they're idle. However, the QNX Neutrino RTOS lets you dynamically change the designated CPU, without having to checkpoint, and then stop and restart the application.

QNX Neutrino supports the concept of hard processor affinity through a *runmask*. Each bit that's set in the runmask represents a processor that a thread can run on. By default, a thread's runmask is set to all ones, allowing it to run on any processor. A value of `0x01` would allow a thread to execute only on the first processor.

By default, a process's or thread's children don't inherit the runmask; there's a separate *inherit mask*.

By careful use of these masks, a systems designer can further optimize the runtime performance of a system (e.g., by relegating nonrealtime processes to a specific processor). In general, however, this shouldn't be necessary, because our realtime scheduler will always preempt a lower-priority thread immediately when a higher-priority thread becomes ready. Processor locking will likely affect only the efficiency of the cache, since threads can be prevented from migrating.

You can specify the runmask for a new thread or process by:

- calling *posix_spawnattr_setrunmask()*, using *posix_spawnattr_setxflags()* to set the POSIX_SPAWN_EXPLICIT_CPU extended flag, and then calling *posix_spawn()*

  Or:

- setting the *runmask* member of the `inheritance` structure and specifying the SPAWN_EXPLICIT_CPU flag when you call *spawn()*

  Or:

- using the `-C` or `-R` option to the `on` utility when you launch a program. This also sets the process's inherit mask to the same value.

You can change the runmask for an existing thread or process by:

- using the _NTO_TCTL_RUNMASK or _NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT command to the *ThreadCtl()* kernel call

  Or:

- using the `-C` or `-R` option to the `slay` utility. If you also use the `-i` option, `slay` sets the inherit mask to the same value.

For more information, see the Multicore Processing chapter of the QNX Neutrino *Programmer's Guide*.

A viable migration strategy

As a midway point between AMP and SMP, BMP offers a viable migration strategy if you wish to move towards full SMP, but you're concerned that your existing code may operate incorrectly in a truly concurrent execution model.

You can port legacy code to a multicore system and initially bind it to a single CPU to ensure correct operation. By judiciously binding applications (and possibly single threads) to specific CPUs, you can isolate potential concurrency issues down to the application and thread level. Resolving these issues will allow the application to run fully concurrently, thereby maximizing the performance gains provided by the multiple processors.

Choosing between AMP, SMP, and BMP

The choice between AMP, SMP, and BMP depends on the problem you're trying to solve:

- AMP works well with legacy applications, but has limited scalability beyond two CPUs.
- SMP offers transparent resource management, but software that hasn't been properly designed for concurrency might have problems.
- BMP offers many of the same benefits as SMP, but guarantees that uniprocessor applications will behave correctly, greatly simplifying the migration of legacy software.

As the following table illustrates, the flexibility to choose from any of these models lets you strike the optimal balance between performance, scalability, and ease of migration.

Feature	SMP	BMP	AMP
Seamless resource sharing	Yes	Yes	—
Scalable beyond dual CPU	Yes	Yes	Limited
Legacy application operation	In most cases	Yes	Yes
Mixed OS environment (e.g., QNX Neutrino and Linux)	—	—	Yes
Dedicated processor by function	—	Yes	Yes
Intercore messaging	Fast (OS primitives)	Fast (OS primitives)	Slower (application)
Thread synchronization between CPUs	Yes	Yes	—
Load balancing	Yes	Yes	—
System-wide debugging and optimization	Yes	Yes	—

Chapter 6 Process Manager

The process manager is capable of creating multiple POSIX processes (each of which may contain multiple POSIX threads).

In the QNX Neutrino RTOS, the microkernel is paired with the Process Manager in a single module (`procnto`). This module is required for all runtime systems. Its main areas of responsibility include:

- process management—manages process creation, destruction, and process attributes such as user ID (*uid*) and group ID (*gid*).
- memory management—manages a range of memory-protection capabilities, shared libraries, and interprocess POSIX shared-memory primitives.
- pathname management—manages the pathname space into which resource managers may attach.

User processes can access microkernel functions directly via kernel calls and process manager functions by sending messages to `procnto`. Note that a user process sends a message by invoking the *MsgSend*()* kernel call.

It's important to note that threads executing within `procnto` invoke the microkernel in exactly the same way as threads in other processes. The fact that the process manager code and the microkernel share the same process address space doesn't imply a “special” or “private” interface. All threads in the system share the same consistent kernel interface and all perform a privilege switch when invoking the microkernel.

Process management

The first responsibility of `procnto` is to dynamically create new processes. These processes will then depend on `procnto`'s other responsibilities of memory management and pathname management.

Process management consists of both process creation and destruction as well as the management of process attributes such as process IDs, process groups, and user IDs.

Process primitives

The process primitives include:

posix_spawn()

POSIX

spawn()

QNX Neutrino

fork()

POSIX

exec*()

POSIX

posix_spawn()

The *posix_spawn()* function creates a child process by directly specifying an executable to load.

To those familiar with UNIX systems, the *posix_spawn()* call is modeled after a *fork()* followed by an *exec*()*. However, it operates much more efficiently in that there's no need to duplicate address spaces as in a *fork()*, only to destroy and replace it when the *exec*()* is called.

In a UNIX system, one of the main advantages of using the *fork()*-then-*exec*()* method of creating a child process is the flexibility in changing the default environment inherited by the new child process. This is done in the forked child just before the *exec*()*. For example, the following simple shell command would close and reopen the standard output before *exec*()*'ing:

```
ls >file
```

You can do the same with *posix_spawn()*; it gives you control over the following classes of environment inheritance, which are often adjusted when creating a new child process:

- file descriptors
- process user and group IDs
- signal mask
- ignored signals
- adaptive partitioning (scheduler) attributes

There's also a companion function, *posix_spawnp()*, that doesn't require the absolute path to the program to spawn, but instead searches for the executable using the caller's *PATH*.

Using the *posix_spawn()* functions is the preferred way to create a new child process.

spawn()

The QNX Neutrino *spawn()* function is similar to *posix_spawn()*.

The *spawn()* function gives you control over the following:

- file descriptors
- process group ID
- signal mask
- ignored signals
- the node to create the process on
- scheduling policy
- scheduling parameters (priority)
- maximum stack size
- runmask (for SMP systems)

The basic forms of the *spawn()* function are:

spawn()

Spawn with the explicitly specified path.

spawnp()

Search the current *PATH* and invoke *spawn()* with the first matching executable.

There's also a set of convenience functions that are built on top of *spawn()* and *spawnp()* as follows:

spawnl()

Spawn with the command line provided as inline arguments.

spawnle()

spawnl() with explicitly passed environment variables.

spawnlp()

spawnp() that follows the command search path.

spawnlpe()

spawnlp() with explicitly passed environment variables.

spawnv()

Spawn with the command line pointed to by an array of pointers.

spawnve()

spawnv() with explicitly passed environment variables.

spawnvp()

spawnv() that follows the command search path.

spawnvpe()

spawnvp() with explicitly passed environment variables.

When a process is *spawn()*'ed, the child process inherits the following attributes of its parent:

- process group ID (unless SPAWN_SETGROUP is set in *inherit.flags*)
- session membership
- real user ID and real group ID
- supplementary group IDs
- priority and scheduling policy
- current working directory and root directory
- file creation mask
- signal mask (unless SPAWN_SETSIGMASK is set in *inherit.flags*)
- signal actions specified as SIG_DFL
- signal actions specified as SIG_IGN (except the ones modified by *inherit.sigdefault* when SPAWN_SETSIGDEF is set in *inherit.flags*)

The child process has several differences from the parent process:

- Signals set to be caught by the parent process are set to the default action (SIG_DFL).
- The child process's *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are tracked separately from the parent's.
- The number of seconds left until a SIGALRM signal would be generated is set to zero for the child process.
- The set of pending signals for the child process is empty.
- File locks set by the parent aren't inherited.
- Per-process timers created by the parent aren't inherited.
- Memory locks and mappings set by the parent aren't inherited.

If the child process is spawned on a remote node, the process group ID and the session membership aren't set; the child process is put into a new session and a new process group.

The child process can access the parent process's environment by using the *environ* global variable (found in **<unistd.h>**).

For more information, see the *spawn()* function in the QNX Neutrino *C Library Reference*.

fork()

The *fork()* function creates a new child process by sharing the same code as the calling process and duplicating the calling process's data to give the child process an exact copy. Most process resources are inherited.

The following resources are explicitly *not* inherited:

- process ID
- parent process ID
- file locks
- pending signals and alarms
- timers

The *fork()* function is typically used for one of two reasons:

- to create a new instance of the current execution environment
- to create a new process running a different program

When creating a new thread, common data is placed in an explicitly created shared memory region. Prior to the POSIX thread standard, this was the only way to accomplish this. With POSIX threads, this use of *fork()* is better accomplished by creating threads within a single process using *pthread_create()*.

When creating a new process running a different program, the call to *fork()* is soon followed by a call to one of the *exec*()* functions. This too is better accomplished by a single call to the *posix_spawn()* function or the QNX Neutrino *spawn()* function, both of which combine both operations with far greater efficiency.

Since QNX Neutrino provides better POSIX solutions than using *fork()*, its use is probably best suited for porting existing code and for writing portable code that must run on a UNIX system that doesn't support the POSIX *pthread_create()* or *posix_spawn()* API.

It's possible to call *fork()* from a multithreaded process, but it can be very difficult to do so safely, so we recommend that you call it only from single-threaded processes. For more information, see “Using *fork()* in a multithreaded process” in the “Processes and Threads” chapter of *Getting Started with QNX Neutrino*.

exec()*

The *exec*()* family of functions replaces the current process with a new process, loaded from an executable file. Since the calling process is replaced, there can be no successful return.

The following *exec*()* functions are defined:

execl()

Exec with the command line provided as inline arguments.

execle()

execl() with explicitly passed environment variables.

execlp()

execl() that follows the command search path.

execlpe()

*execlp()*with explicitly passed environment variables.

execv()

execl() with the command line pointed to by an array of pointers.

execve()

execv() with explicitly passed environment variables.

execvp()

execv() that follows the command search path.

execvpe()

execvp() with explicitly passed environment variables.

The *exec*()* functions usually follow a *fork()* in order to load a new child process. This is better achieved by using the *posix_spawn()* call.

Process loading

Processes loaded from a filesystem using the *exec*()*, *posix_spawn()* or *spawn()* calls are in ELF (Executable and Linking Format).

If the filesystem is on a block-oriented device, the code and data are loaded into main memory. By default, the memory pages containing the binaries are demand-loaded, but you can use the `procnto -m` option to change this; for more information, see "*Locking memory*," later in this chapter.

If the filesystem is memory mapped (e.g., ROM/flash image), the code needn't be loaded into RAM, but may be executed in place. This approach makes all RAM available for data and stack, leaving the code in ROM or flash. In all cases, if the same process is loaded more than once, its code will be shared.

Memory management

While some realtime kernels or executives provide support for memory protection in the development environment, few provide protected memory support for the runtime configuration, citing penalties in memory and performance as reasons. But with memory protection becoming common on many embedded processors, the benefits of memory protection far outweigh the very small penalties in performance for enabling it.

The key advantage gained by adding memory protection to embedded applications, especially for mission-critical systems, is improved *robustness*.

With memory protection, if one of the processes executing in a multitasking environment attempts to access memory that hasn't been explicitly declared or allocated for the type of access attempted, the MMU hardware can notify the OS, which can then abort the thread (at the failing/offending instruction).

This “protects” process address spaces from each other, preventing coding errors in a thread in one process from “damaging” memory used by threads in other processes or even in the OS. This protection is useful both for development and for the installed runtime system, because it makes postmortem analysis possible.

During development, common coding errors (e.g., stray pointers and indexing beyond array bounds) can result in one process/thread accidentally overwriting the data space of another process. If the overwriting touches memory that isn't referenced again until much later, you can spend hours of debugging—often using in-circuit emulators and logic analyzers—in an attempt to find the “guilty party.”

With an MMU enabled, the OS can abort the process the instant the memory-access violation occurs, providing immediate feedback to the programmer instead of mysteriously crashing the system some time later. The OS can then provide the location of the errant instruction in the failed process, or position a symbolic debugger directly on this instruction.

Memory Management Units (MMUs)

A typical MMU operates by dividing physical memory into a number of 4-KB *pages*. The hardware within the processor then uses a set of page tables stored in system memory that define the mapping of virtual addresses (i.e., the memory addresses used within the application program) to the addresses emitted by the CPU to access physical memory.

While the thread executes, the page tables managed by the OS control how the memory addresses that the thread is using are “mapped” onto the physical memory attached to the processor.

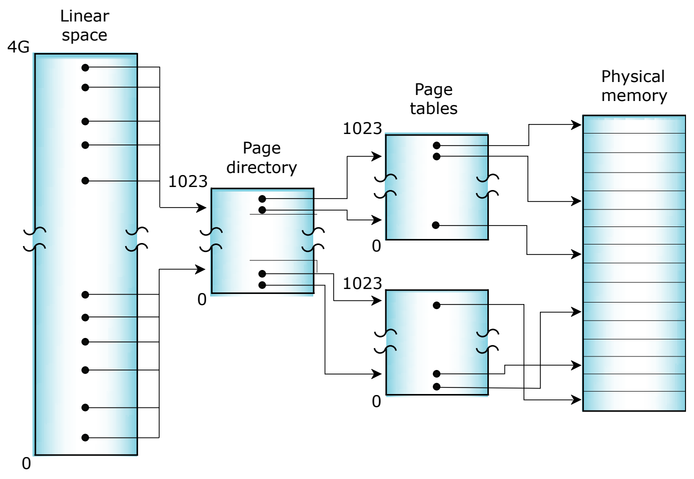


Figure 31: Virtual address mapping (on an x86).

For a large address space with many processes and threads, the number of page-table entries needed to describe these mappings can be significant—more than can be stored within the processor. To maintain performance, the processor caches frequently used portions of the external page tables within a TLB (translation look-aside buffer).

The servicing of "misses" on the TLB cache is part of the overhead imposed by enabling the MMU. Our OS uses various clever page-table arrangements to minimize this overhead.

Associated with these page tables are bits that define the attributes of each page of memory. Pages can be marked as read-only, read-write, etc. Typically, the memory of an executing process would be described with read-only pages for code, and read-write pages for the data and stack.

When the OS performs a context switch (i.e., suspends the execution of one thread and resumes another), it will manipulate the MMU to use a potentially different set of page tables for the newly resumed thread. If the OS is switching between threads *within* a single process, no MMU manipulations are necessary.

When the new thread resumes execution, any addresses generated as the thread runs are mapped to physical memory through the assigned page tables. If the thread tries to use an address not mapped to it, or it tries to use an address in a way that violates the defined attributes (e.g., writing to a read-only page), the CPU will receive a "fault" (similar to a divide-by-zero error), typically implemented as a special type of interrupt.

By examining the instruction pointer pushed on the stack by the interrupt, the OS can determine the address of the instruction that caused the memory-access fault within the thread/process and can act accordingly.

Memory protection at run time

While memory protection is useful during development, it can also provide greater reliability for embedded systems installed in the field. Many embedded systems already employ a hardware "watchdog timer" to detect if the software or hardware has "lost its mind," but this approach lacks the finesse of an MMU-assisted watchdog.

Hardware watchdog timers are usually implemented as a retriggerable monostable timer attached to the processor reset line. If the system software doesn't strobe the hardware timer regularly, the timer will expire and force a processor reset. Typically, some component of the system software will check for system integrity and strobe the timer hardware to indicate the system is "sane."

Although this approach enables recovery from a lockup related to a software or hardware glitch, it results in a complete system restart and perhaps significant "downtime" while this restart occurs.

Software watchdog

When an intermittent software error occurs in a memory-protected system, the OS can catch the event and pass control to a user-written thread instead of the memory dump facilities. This thread can make an intelligent decision about how best to recover from the failure, instead of forcing a full reset as the hardware watchdog timer would do. The software watchdog could:

- Abort the process that failed due to a memory access violation and simply restart that process without shutting down the rest of the system.
- Abort the failed process and any related processes, initialize the hardware to a "safe" state, and then restart the related processes in a coordinated manner.
- If the failure is very critical, perform a coordinated shutdown of the entire system and sound an audible alarm.

The important distinction here is that we retain intelligent, programmed control of the embedded system, even though various processes and threads within the control software may have failed for various reasons. A hardware watchdog timer is still of use to recover from hardware "latch-ups," but for software failures we now have much better control.

While performing some variation of these recovery strategies, the system can also collect information about the nature of the software failure. For example, if the embedded system contains or has access to some mass storage (flash memory, hard drive, a network link to another computer with disk storage), the software watchdog can generate a chronologically archived sequence of dump files. These dump files could then be used for postmortem diagnostics.

Embedded control systems often employ these "partial restart" approaches to surviving intermittent software failures without the operators experiencing any system "downtime" or even being aware of these quick-recovery software failures. Since the dump files are available, the developers of the software can detect and correct software problems without having to deal with the emergencies that result when critical systems fail at inconvenient times. If we compare this to the hardware watchdog timer approach and the prolonged interruptions in service that result, it's obvious what our preference is!

Postmortem dump-file analysis is especially important for mission-critical embedded systems. Whenever a critical system fails in the field, significant effort should be made to identify the cause of the failure so that a "fix" can be engineered and applied to other systems before they experience similar failures.

Dump files give programmers the information they need to fix the problem—without them, programmers may have little more to go on than a customer's cryptic complaint that "the system crashed."

Quality control

By dividing embedded software into a team of cooperating, memory-protected processes (containing threads), we can readily treat these processes as "components" to be used again in new projects. Because of the explicitly defined (and hardware-enforced) interfaces, these processes can be integrated into applications with confidence that they won't disrupt the system's overall reliability. In addition, because the exact binary image (not just the source code) of the process is being reused, we can better control changes and instabilities that might have resulted from recompilation of source code, relinking, new versions of development tools, header files, library routines, etc.

Since the binary image of the process is reused (with its behavior perhaps modified by command-line options), the confidence we have in that binary module from acquired experience in the field more easily carries over to new applications than if the binary image of the process were changed.

As much as we strive to produce error-free code for the systems we deploy, the reality of software-intensive embedded systems is that programming errors will end up in released products. Rather than pretend these bugs don't exist (until the customer calls to report them), we should adopt a “mission-critical” mindset. Systems should be designed to be tolerant of, and able to recover from, software faults. Using the memory protection delivered by integrated MMUs in the embedded systems we build is a good step in that direction.

Full-protection model

Our full-protection model relocates all code in the image into a new virtual space, enabling the MMU hardware and setting up the initial page-table mappings. This allows `procnto` to start in a correct, MMU-enabled environment. The process manager will then take over this environment, changing the mapping tables as needed by the processes it starts.

Private virtual memory

In the full-protection model, each process is given its own private virtual memory. This is accomplished by using the CPU's MMU. The performance cost for a process switch and a message pass will increase due to the increased complexity of obtaining addressability between two completely private address spaces.

Private memory space starts at 0 on x86 and ARM processors. The size depends on the CPU and is 2 or 3.5 gigabytes on 32-bit architectures, and 512 GB on 64-bit architectures; see “Platform-specific limits” in the Understanding System Limits chapter of the QNX Neutrino *User's Guide*.

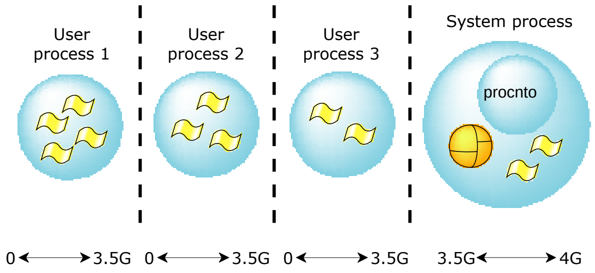


Figure 32: Full protection VM (on an x86).

The memory cost per process may increase by 4 KB to 8 KB for each process's page tables. Note that this memory model supports the POSIX *fork()* call.

Locking memory

The QNX Neutrino RTOS supports POSIX memory locking, so that a process can avoid the latency of fetching a page of memory, by locking the memory so that the page is *memory-resident* (i.e., it remains in physical memory).

The levels of locking are as follows:

Unlocked

Unlocked memory can be paged in and out. Memory is allocated when it's mapped, but page table entries aren't created. The first attempt to access the memory fails, and the thread stays in the WAITPAGE state while the memory manager initializes the memory and creates the page table entries.

Failure to initialize the page results in the receipt of a SIGBUS signal.

Locked

Locked memory may not be paged in or out. Page faults can still occur on access or reference, to maintain usage and modification statistics. Pages that you think are PROT_WRITE are still actually PROT_READ. This is so that, on the first write, the kernel may be alerted that a MAP_PRIVATE page now is different from the shared backing store, and must be privatized.

To lock and unlock a portion of a thread's memory, call *mlock()* and *munlock()*; to lock and unlock *all* of a thread's memory, call *mlockall()* and *munlockall()*. The memory remains locked until the process unlocks it, exits, or calls an *exec*()* function. If the process calls *fork()*, a *posix_spawn*()* function, or a *spawn*()* function, the memory locks are released in the child process.

More than one process can lock the same (or overlapping) region; the memory remains locked until all the processes have unlocked it. Memory locks don't stack; if a process locks the same region more than once, unlocking it once undoes all of the process's locks on that region.

To lock all memory for all applications, specify the `-ml` option for `procnto`. Thus all pages are at least initialized (if still set only to PROT_READ).

Superlocked

(A QNX Neutrino extension) No faulting is allowed at all; all memory must be initialized and privatized, and the permissions set, as soon as the memory is mapped. Superlocking covers the thread's whole address space.

To superlock memory, obtain I/O privileges by:

1. Enabling the PROCMGR_AID_IO ability for the process. For more information, see *procmgr_ability()*.
2. Calling *ThreadCtl()*, specifying the _NTO_TCTL_IO flag:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

To superlock all memory for all applications, specify the `-mL` option for `procnto`.

For MAP_LAZY mappings, memory isn't allocated or mapped until the memory is first referenced for any of the above types. Once it's been referenced, it obeys the above rules—it's a programmer error to touch a MAP_LAZY area in a critical region (where interrupts are disabled or in an ISR) that hasn't already been referenced.

Pathname management

I/O resources *aren't* built into the microkernel, but are instead provided by resource manager processes that may be started dynamically at runtime. The `procnto` manager allows resource managers, through a standard API, to adopt a subset of the pathname space as a "domain of authority" to administer.

As other resource managers adopt their respective domains of authority, `procnto` becomes responsible for maintaining a pathname tree to track the processes that own portions of the pathname space. An adopted pathname is sometimes referred to as a "prefix" because it prefixes any pathnames that lie beneath it; prefixes can be arranged in a hierarchy called a *prefix tree*. The adopted pathname is also called a *mountpoint*, because that's where a server mounts into the pathname.

This approach to pathname space management is what allows QNX Neutrino to preserve the POSIX semantics for device and file access, while making the presence of those services optional for small embedded systems.

At startup, `procnto` populates the pathname space with the following pathname prefixes:

Prefix	Description
/	Root of the filesystem.
/proc/boot/	Some of the files from the boot image presented as a flat filesystem.
/proc/*pid*	The running processes, each represented by its process ID (PID). For more information, see "Controlling processes via the **/proc** filesystem" in the Processes chapter of the QNX Neutrino *Programmer's Guide*.
/dev/zero	A device that always returns zero. Used for allocating zero-filled pages using the *mmap()* function.
/dev/mem	A device that represents all physical memory.

Resolving pathnames

When a process opens a file, the POSIX-compliant *open()* library routine first sends the pathname to `procnto`, where the pathname is compared against the prefix tree to determine which resource managers should be sent the *open()* message.

The prefix tree may contain identical or partially overlapping regions of authority—multiple servers can register the same prefix. If the regions are identical, the order of resolution can be specified (see "Ordering mountpoints," below). If the regions are overlapping, the responses from the path manager are ordered with the longest prefixes first; for prefixes of equal length, the same specified order of resolution applies as for identical regions.

For example, suppose we have these prefixes registered:

Prefix	Description
/	Power-Safe filesystem (`fs-qnx6.so`)
/dev/ser1	Serial device manager (`devc-ser*`)
/dev/ser2	Serial device manager (`devc-ser*`)
/dev/hd0	Raw disk volume (`devb-eide`)

The filesystem manager has registered a prefix for a mounted Power-Safe filesystem (i.e., **/**). The block device driver has registered a prefix for a block special file that represents an entire physical hard drive (i.e., **/dev/hd0**). The serial device manager has registered two prefixes for the two PC serial ports.

The following table illustrates the longest-match rule for pathname resolution:

This pathname:	matches:	and resolves to:
/dev/ser1	**/dev/ser1**	`devc-ser*`
/dev/ser2	**/dev/ser2**	`devc-ser*`
/dev/ser	**/**	`fs-qnx6.so`
/dev/hd0	**/dev/hd0**	`devb-eide.so`
/usr/jhsmith/test	**/**	`fs-qnx6.so`

Ordering mountpoints

Generally the order of resolving a filename is the order in which you mounted the filesystems at the same mountpoint (i.e., new mounts go on top of or in front of any existing ones). You can specify the order of resolution when you mount the filesystem. For example, you can use:

- the `before` and `after` keywords for block I/O (`devb-*`) drivers, in the `blk` options
- the `-Z b` and `-Z a` options to `fs-cifs`, `fs-nfs2`, and `fs-nfs3`

You can also use the `-o` option to `mount` with these keywords:

`before`

Mount the filesystem so that it's resolved before any other filesystems mounted at the same pathname (in other words, it's placed in front of any existing mount). When you access a file, the system looks on this filesystem first.

`after`

Mount the filesystem so that it's resolved after any other filesystems mounted at the same pathname (in other words, it's placed behind any existing mounts). When you access a file, the system looks on this filesystem last, and only if the file wasn't found on any other filesystems.

If you specify the appropriate `before` option, the filesystem floats in front of any other filesystems mounted at the same mountpoint, except those that you later mount with `before`. If you specify `after`, the filesystem goes behind any other filesystems mounted at the same mountpoint, except those that are already mounted with `after`. So, the search order for these filesystems is:

1. those mounted with `before`
2. those mounted with no flags
3. those mounted with `after`

with each list searched in order of mount requests. The first server to claim the name gets it. You would typically use `after` to have a filesystem wait at the back and pick up things the no one else is handling, and `before` to make sure a filesystem looks first at filenames.

Single-device mountpoints

Consider an example involving three servers:

Server A

A Power-Safe filesystem. Its mountpoint is **/**. It contains the files **bin/true** and **bin/false**.

Server B

A flash filesystem. Its mountpoint is **/bin**. It contains the files **ls** and **echo**.

Server C

A single device that generates numbers. Its mountpoint is **/dev/random**.

At this point, the process manager's internal mount table would look like this:

Mountpoint	Server
/	Server A (Power-Safe filesystem)
/bin	Server B (flash filesystem)
/dev/random	Server C (device)

Of course, each “Server” name is actually an abbreviation for the *nd, pid, chid* for that particular server channel.

Now suppose a client wants to send a message to Server C. The client's code might look like this:

```
int fd;
fd = open("/dev/random", ...);
read(fd, ...);
close(fd);
```

In this case, the C library will ask the process manager for the servers that could *potentially* handle the path **/dev/random**. The process manager would return a list of servers:

- Server C (most likely; longest path match)
- Server A (least likely; shortest path match)

From this information, the library will then contact each server in turn and send it an *open* message, including the component of the path that the server should validate:

1. Server C receives a null path, since the request came in on the same path as the mountpoint.
2. Server A receives the path **dev/random**, since its mountpoint was **/**.

As soon as one server positively acknowledges the request, the library won't contact the remaining servers. This means Server A is contacted only if Server C denies the request.

This process is fairly straightforward with single device entries, where the first server is generally the server that will handle the request. Where it becomes interesting is in the case of *unioned filesystem mountpoints*.

Unioned filesystem mountpoints

Let's assume we have two servers set up as before:

Server A

A Power-Safe filesystem. Its mountpoint is **/**. It contains the files **bin/true** and **bin/false**.

Server B

A flash filesystem. Its mountpoint is **/bin**. It contains the files **ls** and **echo**.

Note that each server has a **/bin** directory, but with different contents.

Once both servers are mounted, you would see the following due to the union of the mountpoints:

/

Server A

/bin

Servers A and B

/bin/echo

Server B

/bin/false

Server A

/bin/ls

Server B

/bin/true

Server A

What's happening here is that the resolution for the path **/bin** takes place as before, but rather than limit the return to just one connection ID, *all* the servers are contacted and asked about their handling for the path:

```
DIR *dirp;
dirp = opendir("/bin", ...);
closedir(dirp);
```

which results in:

1. Server B receives a null path, since the request came in on the same path as the mountpoint.
2. Server A receives the path **"bin"**, since its mountpoint was **"/"**.

The result now is that we have a collection of file descriptors to servers who handle the path **/bin** (in this case two servers); the actual directory name entries are read in turn when a *readdir()* is called. If any of the names in the directory are accessed with a regular open, then the normal resolution procedure takes place and only one server is accessed.

For a more detailed look at this, see "Unioned filesystems" in the Resource Managers chapter of *Getting Started with QNX Neutrino*.

Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

Why overlay mountpoints?

This overlaying of mountpoints is a very handy feature when doing field updates, servicing, etc. It also makes for a more unified system, where pathnames result in connections to servers regardless of what services they're providing, thus resulting in a more unified API.

Symbolic prefixes

We've discussed prefixes that map to a resource manager. A second form of prefix, known as a *symbolic prefix*, is a simple string substitution for a matched prefix.

You create symbolic prefixes using the POSIX `ln` (link) command. This command is typically used to create hard or symbolic links on a filesystem by using the `-s` option. If you also specify the `-P` option, then a symbolic link is created in the in-memory prefix space of `procnto`.

Command	Description
`ln -s` *existing_file symbolic_link*	Create a filesystem symbolic link.
`ln -Ps` *existing_file symbolic_link*	Create a prefix tree symbolic link.

Note that a prefix tree symbolic link will always take precedence over a filesystem symbolic link.

For example, assume you're running on a machine that doesn't have a local filesystem. However, there's a filesystem on another node (say `neutron`) that you wish to access as "**/bin**". You accomplish this using the following symbolic prefix:

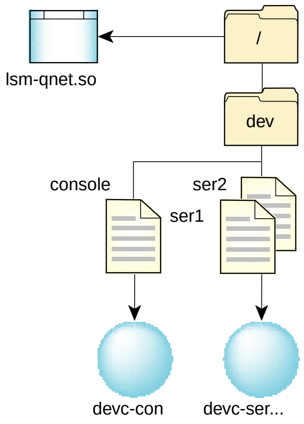
```
ln -Ps /net/neutron/bin  /bin
```

This will cause **/bin** to be mapped into **/net/neutron/bin**. For example, **/bin/ls** will be replaced with the following:

/net/neutron/bin/ls

This new pathname will again be applied against the prefix tree, but this time the prefix matched will be **/net**, which will point to `lsm-qnet`. The `lsm-qnet` resource manager will then resolve the **neutron** component, and redirect further resolution requests to the node called `neutron`. On node `neutron`, the rest of the pathname (i.e. **/bin/ls**) will be resolved against the prefix space on that node. This will resolve to the filesystem manager on node `neutron`, where the *open()* request will be directed. With just a few characters, this symbolic prefix has allowed us to access a remote filesystem as though it were local.

It's not necessary to run a local filesystem process to perform the redirection. A diskless workstation's prefix tree might look something like this:



With this prefix tree, local devices such as **/dev/ser1** or **/dev/console** will be routed to the local character device manager, while requests for other pathnames will be routed to the remote filesystem.

Creating special device names

You can also use symbolic prefixes to create special device names. For example, if your modem was on **/dev/ser1**, you could create a symbolic prefix of **/dev/modem** as follows:

```
ln -Ps /dev/ser1 /dev/modem
```

Any request to open **/dev/modem** will be replaced with **/dev/ser1**. This mapping would allow the modem to be changed to a different serial port simply by changing the symbolic prefix and without affecting any applications.

Relative pathnames

Pathnames need not start with slash. In such cases, the path is considered *relative* to the current working directory.

The OS maintains the current working directory as a character string. Relative pathnames are always converted to full network pathnames by prepending the current working directory string to the relative pathname.

Note that different behaviors result when your current working directory starts with a slash versus starting with a network root.

Network root

If the current working directory begins with a *network root* in the form **/net/***node_name*, it's said to be *specific* and locked to the pathname space of the specified node. If you don't specify a network root, the default one is prepended.

For example, this command:

```
cd /net/percy
```

is an example of the first (specific) form, and would lock future relative pathname evaluation to be on node `percy`, no matter what your default network root happens to be. Subsequently entering `cd dev` would put you in **/net/percy/dev**.

On the other hand, this command:

```
cd /
```

would be of the second form, where the default network root would affect the relative pathname resolution. For example, if your default network root were **/net/florence**, then entering `cd dev` would put you in **/net/florence/dev**. Since the current working directory doesn't start with a node override, the default network root is prepended to create a fully specified network pathname.

To run a command with a specific network root, use the `on` command, specifying the `-f` option:

`on -f /net/percy` *command*

This runs the given command with **/net/percy** as the network root; that is, it searches for the command—and any files with relative paths specified as arguments—on **/net/percy** and runs the command on **/net/percy**. In contrast, this:

`on -n /net/percy` *command*

searches for the given command—and any files with relative paths—on your *local* node and runs the command on **/net/percy**.

In a program, you can specify a network root when you call *chroot()*.

This really isn't as complicated as it may seem. Most of the time, you don't specify a network root, and everything you do will simply work within your namespace (defined by your default network root). Most users will log in, accept the normal default network root (i.e., the namespace of their own node), and work within that environment.

File descriptor namespace

Once an I/O resource has been opened, a different namespace comes into play. The *open()* returns an integer referred to as a *file descriptor* (FD), which is used to direct all further I/O requests to that resource manager.

Unlike the pathname space, the file descriptor namespace is completely local to each process. The resource manager uses the combination of a SCOID (server connection ID) and FD (file descriptor/connection ID) to identify the control structure associated with the previous *open()* call. This structure is referred to as an *open control block* (OCB) and is contained within the resource manager.

The following diagram shows an I/O manager taking some SCOID, FD pairs and mapping them to OCBs.

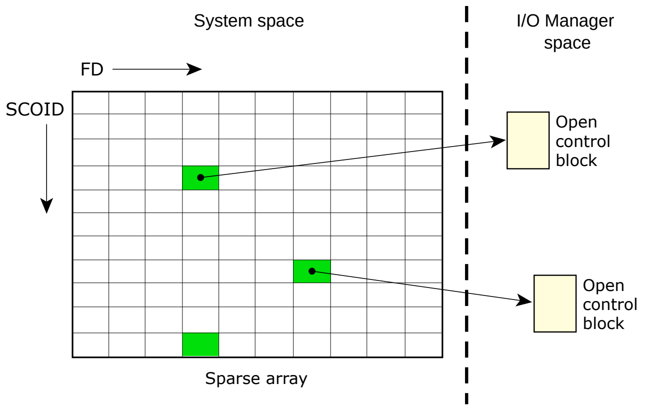


Figure 33: The SCOID and FD map to an OCB of an I/O Manager.

Open control blocks

The open control block (OCB) contains active information about the open resource.

For example, the filesystem keeps the current seek point within the file here. Each *open()* creates a new OCB. Therefore, if a process opens the same file twice, any calls to *lseek()* using one FD will not affect the seek point of the other FD. The same is true for different processes opening the same file.

The following diagram shows two processes, in which one opens the same file twice, and the other opens it once. There are no shared FDs.

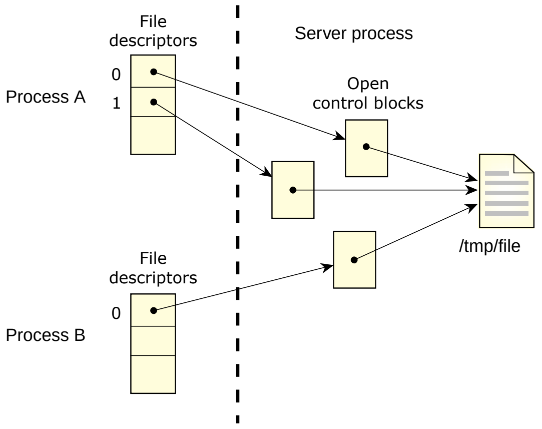


Figure 34: Two processes open the same file.

FDs are a *process* resource, not a thread resource.

Several file descriptors in one or more processes can refer to the same OCB. This is accomplished by two means:

- A process may use the *dup()*, *dup2()*, or *fcntl()* functions to create a duplicate file descriptor that refers to the same OCB.

- When a new process is created via *fork()*, *posix_spawn()*, or *spawn()*, all open file descriptors are by default inherited by the new process; these inherited descriptors refer to the same OCBs as the corresponding file descriptors in the parent process.

When several FDs refer to the same OCB, then any change in the state of the OCB is immediately seen by all processes that have file descriptors linked to the same OCB.

For example, if one process uses the *lseek()* function to change the position of the seek point, then reading or writing takes place from the new position no matter which linked file descriptor is used.

The following diagram shows two processes in which one opens a file twice, then does a *dup()* to get a third FD. The process then creates a child that inherits all open files.

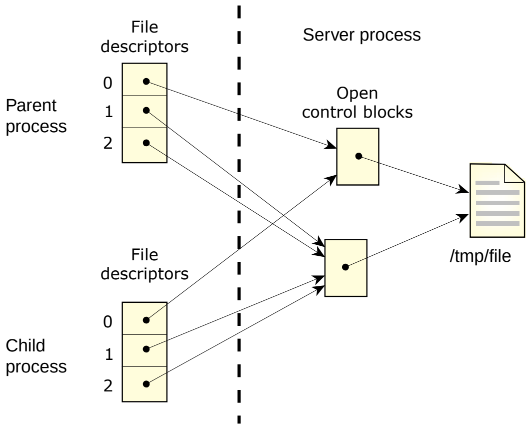


Figure 35: A process opens a file twice.

You can prevent a file descriptor from being inherited when you *posix_spawn()*, *spawn()*, or *exec*()* by calling the *fcntl()* function and setting the FD_CLOEXEC flag.

Chapter 7
Dynamic Linking

In a typical system, a number of programs will be running. Each program relies on a number of functions, some of which will be "standard" C library functions, like *printf()*, *malloc()*, *write()*, etc.

If every program uses the standard C library, it follows that each program would normally have a unique copy of this particular library present within it. Unfortunately, this results in wasted resources. Since the C library is common, it makes more sense to have each program *reference* the common instance of that library, instead of having each program *contain* a copy of the library. This approach yields several advantages, not the least of which is the savings in terms of total system memory required.

Before we go any further, let's look at some terminology:

Linker

A tool, such as `ld`, that you typically run just after compiling your program, in order to combine object and archive files, relocate their data, and resolve symbol references.

Runtime linker

A tool that finds and loads shared objects when you run your program. This is also known as a *dynamic linker*, but we'll use *runtime linker* to avoid any confusion with dynamic linking, which the (non-runtime) linker does.

The name of the runtime linker is `ldd` (which is also the name of a utility that lists the shared objects that a program requires). In the `.interp` section of an ELF file, it's called **ldqnx.so** for 32-bit targets, and **ldqnx-64.so** for 64-bit targets. It's in **libc**, which is why you need to create the appropriate symbolic link for it in your `mkifs` buildfiles:

```
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
```

or:

```
[type=link] /usr/lib/ldqnx-64.so.2=/proc/boot/libc.so
```

Statically linked

The program and the particular library that it's linked against are combined by the linker at link time.

This means that the binding between the program and the particular library is fixed and known at link time—well in advance of the program's ever running. It also means that we can't change this binding, unless we relink the program with a new version of the library.

You might consider linking a program statically in cases where you weren't sure whether the correct version of a library will be available at runtime, or if you were testing a new version of a library that you don't yet want to install as shared.

Programs that are linked statically are linked against archives of objects (*libraries*) that typically have the extension of **.a**. An example of such a collection of objects is the standard C library, **libc.a**.

Dynamically linked

The program and the particular library it references *aren't* combined by the linker at link time.

Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references. This means that the binding between the program and the shared object is done *at runtime*—before the program starts, the appropriate shared objects are found and bound.

This type of program is called a *partially bound executable*, because it isn't fully *resolved*—the linker, at link time, didn't cause all the referenced symbols in the program to be associated with specific code from the library. Instead, the linker simply said: "This program calls some functions within a particular shared object, so I'll just make a note of *which* shared object these functions are in, and continue on." Effectively, this defers the binding until runtime.

Programs that are linked dynamically are linked against shared objects that have the extension **.so**. An example of such an object is the shared object version of the standard C library, **libc.so**.

You use a command-line option to the compiler driver `qcc` to tell the tool chain whether you're linking statically or dynamically. This command-line option then determines the extension used (either **.a** or **.so**).

Augmenting code at runtime

Taking this one step further, a program may not know which functions it needs to call until it's running. While this may *seem* a little strange initially (after all, how could a program *not* know what functions it's going to call?), it really can be a very powerful feature. Here's why.

Consider a "generic" disk driver. It starts, probes the hardware, and detects a hard disk. The driver would then dynamically load the **io-blk** code to handle the disk blocks, because it found a block-oriented device. Now that the driver has access to the disk at the block level, it finds two partitions present on the disk: a DOS partition and a Power-Safe partition. Rather than force the disk driver to contain filesystem drivers for all possible partition types it may encounter, we kept it simple: it doesn't have *any* filesystem drivers! At runtime, it detects the two partitions and *then* knows that it should load the `fs-dos.so` and `fs-qnx6.so` filesystem code to handle those partitions.

By deferring the decision of which functions to call, we've enhanced the flexibility of the disk driver (and also reduced its size).

How shared objects are used

To understand how a program makes use of shared objects, let's first see the format of an executable and then examine the steps that occur when the program starts.

ELF format

The QNX Neutrino RTOS uses the ELF (Executable and Linking Format) binary format, which is currently used in SVR4 Unix systems. ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at runtime.

In the following diagram, we show two views of an ELF file: the linking view and the execution view. The linking view, which is used when the program or library is linked, deals with *sections* within an object file. Sections contain the bulk of the object file information: data, instructions, relocation information, symbols, debugging information, etc. The execution view, which is used when the program runs, deals with *segments*.

At link time, the program or library is built by merging together sections with similar attributes into segments. Typically, all the executable and read-only data sections are combined into a single "`text`" segment, while the data and "`BSS`"s are combined into the "`data`" segment. These segments are called *load segments*, because they need to be loaded in memory at process creation. Other sections such as symbol information and debugging sections are merged into other, nonload segments.

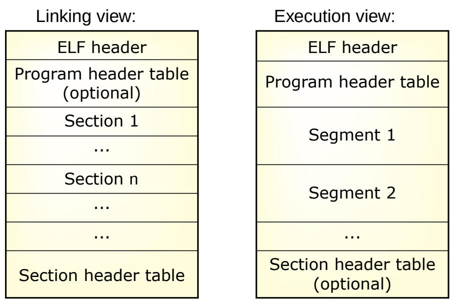


Figure 36: Object file format: linking view and execution view.

ELF without COFF

Most implementations of ELF loaders are derived from *COFF* (Common Object File Format) loaders; they use the linking view of the ELF objects at load time. This is inefficient because the program loader must load the executable using sections. A typical program could contain a large number of sections, each of which would have to be located in the program and loaded into memory separately.

QNX Neutrino, however, doesn't rely at all on the COFF technique of loading sections. When developing our ELF implementation, we worked directly from the ELF spec and kept efficiency paramount. The ELF loader uses the "execution view" of the program. By using the execution view, the task of the loader is greatly simplified: all it has to do is copy to memory the load segments (usually two) of the program or library. As a result, process creation and library loading operations are much faster.

Memory layout for a typical process

The diagram below shows the memory layout of a typical process. The process load segments (corresponding to “`text`” and“`data`” in the diagram) are loaded at the process's base address. The main stack is located just below and grows downwards. Any additional threads that are created will have their own stacks, located below the main stack. Each of the stacks is separated by a guard page to detect stack overflows. The heap is located above the process and grows upwards.

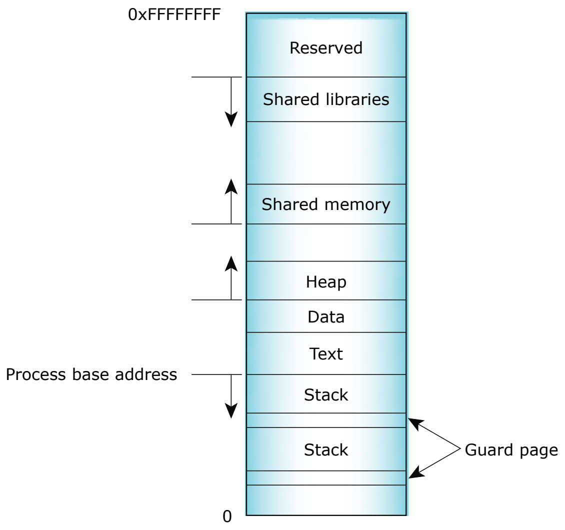


Figure 37: Process memory layout on an x86.

In the middle of the process's address space, a large region is reserved for shared objects. Shared libraries are located at the top of the address space and grow downwards.

When a new process is created, the process manager first maps the two segments from the executable into memory. It then decodes the program's ELF header. If the program header indicates that the executable was linked against a shared library, the process manager will extract the name of the *dynamic interpreter* from the program header. The dynamic interpreter points to a shared library that contains the *runtime linker* code. The process manager will load this shared library in memory and will then pass control to the runtime linker code in this library.

Runtime linker

The runtime linker is invoked when a program that was linked against a shared object is started or when a program requests that a shared object be dynamically loaded. The runtime linker is contained within the C runtime library.

The runtime linker performs several tasks when loading a shared library (**.so** file):

1. If the requested shared library isn't already loaded in memory, the runtime linker loads it:
    - If the shared library name is fully qualified (i.e., begins with a slash), it's loaded directly from the specified location. If it can't be found there, no further searches are performed.
    - If it's not a fully qualified pathname, the runtime linker searches for it as follows:

        a. If the executable's dynamic section contains a `DT_RPATH` tag, then the path specified by `DT_RPATH` is searched.

  b. If the shared library isn't found, the runtime linker searches for it in the directories specified by *LD_LIBRARY_PATH*.

  For security reasons, the runtime linker unsets *LD_LIBRARY_PATH* if the binary has the setuid bit set.

  c. If the shared library still isn't found, then the runtime linker searches for the default library search path as specified by the *LD_LIBRARY_PATH* environment variable to `procnto` (i.e., the CS_LIBPATH configuration string). If none has been specified, then the default library path is set to the image filesystem's path.

2. Once the requested shared library is found, it's loaded into memory. For ELF shared libraries, this is a very efficient operation: the runtime linker simply needs to use the *mmap()* call twice to map the two load segments into memory.
3. The shared library is then added to the internal list of all libraries that the process has loaded. The runtime linker maintains this list.
4. The runtime linker then decodes the dynamic section of the shared object.

This dynamic section provides information to the linker about other libraries that this library was linked against. It also gives information about the relocations that need to be applied and the external symbols that need to be resolved. The runtime linker will first load any other required shared libraries (which may themselves reference other shared libraries). It will then process the relocations for each library. Some of these relocations are local to the library, while others require the runtime linker to resolve a global symbol. In the latter case, the runtime linker will search through the list of libraries for this symbol. In ELF files, hash tables are used for the symbol lookup, so they're very fast. The order in which libraries are searched for symbols is very important, as we'll see in the section on *"Symbol name resolution"* below.

Once all relocations have been applied, any initialization functions that have been registered in the shared library's init section are called. This is used in some implementations of C++ to call global constructors.

Loading a shared library at runtime

A process can load a shared library at runtime by using the *dlopen()* call, which instructs the runtime linker to load this library. Once the library is loaded, the program can call any function within that library by using the *dlsym()* call to determine its address.

Remember: shared libraries are available only to processes that are *dynamically linked.*

The program can also determine the symbol associated with a given address by using the *dladdr()* call. Finally, when the process no longer needs the shared library, it can call *dlclose()* to unload the library from memory.

Symbol name resolution

When the runtime linker loads a shared library, the symbols within that library have to be resolved. The order and the scope of the symbol resolution are important. If a shared library calls a function that happens to exist by the same name in several libraries that the program has loaded, the order in which these libraries are searched for this symbol is critical. This is why the OS defines several options that can be used when loading libraries.

All the objects (executables and libraries) that have global scope are stored on an internal list (the *global list*). Any global-scope object, by default, makes available all of its symbols to any shared library that gets loaded. The global list initially contains the executable and any libraries that are loaded at the program's startup.

By default, when a new shared library is loaded by using the *dlopen()* call, symbols within that library are resolved by searching in this order through:

1. the shared library
2. the list of libraries specified by the *LD_PRELOAD* environment variable. You can use this environment variable to add or change functionality when you run a program.

   For security reasons, the runtime linker unsets *LD_PRELOAD* if the binary has the setuid bit set.

3. the global list
4. any dependent objects that the shared library references (i.e., any other libraries that the shared library was linked against)

The runtime linker's scoping behavior can be changed in two ways when *dlopen()*'ing a shared library:

- When the program loads a new library, it may instruct the runtime linker to place the library's symbols on the global list by passing the RTLD_GLOBAL flag to the *dlopen()* call. This will make the library's symbols available to any libraries that are subsequently loaded.
- The list of objects that are searched when resolving the symbols within the shared library can be modified. If the RTLD_GROUP flag is passed to *dlopen()*, then only objects that the library directly references will be searched for symbols. If the RTLD_WORLD flag is passed, only the objects on the global list will be searched.

Chapter 8 Resource Managers

To give the QNX Neutrino RTOS a great degree of flexibility, to minimize the runtime memory requirements of the final system, and to cope with the wide variety of devices that may be found in a custom embedded system, the OS allows user-written processes to act as *resource managers* that can be started and stopped dynamically.

Resource managers are typically responsible for presenting an interface to various types of devices. This may involve managing actual hardware devices (like serial ports, parallel ports, network cards, and disk drives) or virtual devices (like **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers.* But unlike device drivers, resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program.

What is a resource manager?

Since the QNX Neutrino RTOS is a distributed, microkernel OS with virtually all nonkernel functionality provided by user-installable programs, a clean and well-defined interface is required between client programs and resource managers. All resource manager functions are documented; there's no "magic" or private interface between the kernel and a resource manager.

In fact, a resource manager is basically a user-level server program that accepts messages from other programs and, optionally, communicates with hardware. Again, the power and flexibility of our native IPC services allow the resource manager to be decoupled from the OS.

The binding between the resource manager and the client programs that use the associated resource is done through a flexible mechanism called *pathname space mapping*.

In pathname space mapping, an association is made between a pathname and a resource manager. The resource manager sets up this pathname space mapping by informing the process manager that it is the one responsible for handling requests at (or below, in the case of filesystems), a certain *mountpoint*. This allows the process manager to associate services (i.e., functions provided by resource managers) with pathnames.

For example, a serial port may be managed by a resource manager called `devc-ser*`, but the actual resource may be called **/dev/ser1** in the pathname space. Therefore, when a program requests serial port services, it typically does so by opening a serial port—in this case **/dev/ser1**.

Why write a resource manager?

Here are a few reasons why you may want to write a resource manager:

- The client API is POSIX.

  The API for communicating with the resource manager is for the most part, POSIX. All C programmers are familiar with the *open()*, *read()*, and *write()* functions. Training costs are minimized, and so is the need to document the interface to your server.

- You can reduce the number of interface types.

  If you have many server processes, writing each server as a resource manager keeps the number of different interfaces that clients need to use to a minimum.

  For example, suppose you have a team of programmers building your overall application, and each programmer is writing one or more servers for that application. These programmers may work directly for your company, or they may belong to partner companies who are developing addon hardware for your modular platform.

  If the servers are resource managers, then the interface to all of those servers is the POSIX functions: *open()*, *read()*, *write()*, and whatever else makes sense. For control-type messages that don't fit into a read/write model, there's *devctl()* (although *devctl()* isn't POSIX).

- Command-line utilities can communicate with resource managers.

  Since the API for communicating with a resource manager is the POSIX set of functions, and since standard POSIX utilities use this API, you can use the utilities for communicating with the resource managers.

For instance, suppose a resource manager registers the name **/proc/my_stats**. If you open this name and read from it, the resource manager responds with a body of text that describes its statistics.

The `cat` utility takes the name of a file and opens the file, reads from it, and displays whatever it reads to standard output (typically the screen). As a result, you could type:

```
cat /proc/my_stats
```

and the resource manager would respond with the appropriate statistics.

You could also use command-line utilities for a robot-arm driver. The driver could register the name, **/dev/robot/arm/angle**, and interpret any writes to this device as the angle to set the robot arm to. To test the driver from the command line, you'd type:

```
echo 87 >/dev/robot/arm/angle
```

The `echo` utility opens **/dev/robot/arm/angle** and writes the string ("87") to it. The driver handles the write by setting the robot arm to 87 degrees. Note that this was accomplished without writing a special tester program.

Another example would be names such as **/dev/robot/registers/r1**, **r2**, and so on. Reading from these names returns the contents of the corresponding registers; writing to these names sets the corresponding registers to the given values.

Even if all of your other IPC is done via some non-POSIX API, it's still worth having one thread written as a resource manager for responding to reads and writes for doing things as shown above.

The types of resource managers

Depending on how much work you want to do yourself in order to present a proper POSIX filesystem to the client, you can break resource managers into two types:

- *Device resource managers*
- *Filesystem resource managers*

Device resource managers

Device resource managers create only single-file entries in the filesystem, each of which is registered with the process manager. Each name usually represents a single device. These resource managers typically rely on the resource-manager library to do most of the work in presenting a POSIX device to the user.

For example, a serial port driver registers names such as `/dev/ser1` and `/dev/ser2`. When the user does `ls -l /dev`, the library does the necessary handling to respond to the resulting _IO_STAT messages with the proper information. The person who writes the serial port driver can concentrate instead on the details of managing the serial port hardware.

Filesystem resource managers

Filesystem resource managers register a *mountpoint* with the process manager. A mountpoint is the portion of the path that's registered with the process manager. The remaining parts of the path are managed by the filesystem resource manager. For example, when a filesystem resource manager attaches a mountpoint at **/mount**, and the path **/mount/home/thomasf** is examined:

/mount/

Identifies the mountpoint that's managed by the process manager.

home/thomasf

Identifies the remaining part that's to be managed by the filesystem resource manager.

Here are some examples of using filesystem resource managers:

- flash filesystem drivers (although the source code for flash drivers takes care of these details)
- a `tar` filesystem process that presents the contents of a `tar` file as a filesystem that the user can `cd` into and `ls` from
- a mailbox-management process that registers the name **/mailboxes** and manages individual mailboxes that look like directories, and files that contain the actual messages.

Communication via native IPC

Once a resource manager has established its pathname prefix, it will receive messages whenever any client program tries to do an *open()*, *read()*, *write()*, etc. on that pathname.

For example, after `devc-ser*` has taken over the pathname **/dev/ser1**, and a client program executes:

```
fd = open ("/dev/ser1", O_RDONLY);
```

the client's C library will construct an `io_open` message, which it then sends to the `devc-ser*` resource manager via IPC.

Some time later, when the client program executes:

```
read (fd, buf, BUFSIZ);
```

the client's C library constructs an `io_read` message, which is then sent to the resource manager.

A key point is that *all* communications between the client program and the resource manager are done through *native IPC messaging*. This allows for a number of unique features:

- A well-defined interface to application programs. In a development environment, this allows a very clean division of labor for the implementation of the client side and the resource manager side.
- A simple interface to the resource manager. Since all interactions with the resource manager go through native IPC, and there are no special “back door” hooks or arrangements with the OS, the writer of a resource manager can focus on the task at hand, rather than worry about all the special considerations needed in other operating systems.
- Free network transparency. Since the underlying native IPC messaging mechanism is inherently network-distributed without any additional effort required by the client or server (resource manager), programs can seamlessly access resources on other nodes in the network without even being aware that they're going over a network.

All QNX Neutrino device drivers and filesystems are implemented as resource managers. This means that everything that a “native” QNX Neutrino device driver or filesystem can do, a user-written resource manager can do as well.

Consider FTP filesystems, for instance. Here a resource manager would take over a portion of the pathname space (e.g., **/ftp**) and allow users to `cd` into FTP sites to get files. For example, `cd /ftp/rtfm.mit.edu/pub` would connect to the FTP site **rtfm.mit.edu** and change directory to **/pub**. After that point, the user could open, edit, or copy files.

Application-specific filesystems would be another example of a user-written resource manager. Given an application that makes extensive use of disk-based files, a custom tailored filesystem can be written that works with that application and delivers superior performance.

The possibilities for custom resource managers are limited only by the application developer's imagination.

Resource manager architecture

Here is the heart of a resource manager:

```
initialize the dispatch interface
register the pathname with the process manager
DO forever
    receive a message
    SWITCH on the type of message
        CASE io_open:
            perform io_open processing
            ENDCASE
        CASE io_read:
            perform io_read processing
            ENDCASE
        CASE io_write:
            perform io_write processing
            ENDCASE
        .   // etc. handle all other messages
        .   // that may occur, performing
        .   // processing as appropriate
    ENDSWITCH
ENDDO
```

The architecture contains three parts:

1. A channel is created so that client programs can connect to the resource manager to send it messages.
2. The pathname (or pathnames) that the resource manager is going to be responsible for is registered with the process manager, so that it can resolve open requests for that particular pathname to this resource manager.
3. Messages are received and processed.

This message-processing structure (the switch/case, above) is required for each and every resource manager. However, we provide a set of convenient library functions to handle this functionality (and other key functionality as well).

Message types

Architecturally, there are two categories of messages that a resource manager will receive:

- *connect messages*
- *I/O messages*

A connect message is issued by the client to perform an operation based on a pathname (e.g., an `io_open` message). This may involve performing operations such as permission checks (does the client have the correct permission to open this device?) and setting up a *context* for that request.

An I/O message is one that relies upon this context (created between the client and the resource manager) to perform subsequent processing of I/O messages (e.g., `io_read`).

There are good reasons for this design. It would be inefficient to pass the full pathname for each and every *read()* request, for example. The `io_open` handler can also perform tasks that we want done only once (e.g., permission checks), rather than with each I/O message. Also, when the *read()* has read 4096 bytes from a disk file, there may be another 20 megabytes still waiting to be read. Therefore, the *read()* function would need to have some context information telling it the position within the file it's reading from.

The resource manager shared library

In a custom embedded system, part of the design effort may be spent writing a resource manager, because there may not be an off-the-shelf driver available for the custom hardware component in the system.

Our resource manager shared library makes this task relatively simple.

Automatic default message handling

If there are functions that the resource manager doesn't want to handle for some reason (e.g., a digital-to-analog converter doesn't support a function such as *lseek()*, or the software doesn't require it), the shared library will conveniently supply default actions.

There are two levels of default actions:

- The first level simply returns ENOSYS to the client application, informing it that that particular function is not supported.
- The second level (i.e., the *iofunc_*()* shared library) allows a resource manager to automatically handle various functions.

For more information on default actions, see the section on "*Second-level default message handling*" in this chapter.

open(), *dup()*, and *close()*

Another convenient service that the resource manager shared library provides is the automatic handling of *dup()* messages.

Suppose that the client program executed code that eventually ended up performing:

```
fd = open ("/dev/device", O_RDONLY);
...
fd2 = dup (fd);
...
fd3 = dup (fd);
...
close (fd3);
...
close (fd2);
```

```
...
close (fd);
```

The client would generate an `io_open` message for the first *open()*, and then two `io_dup` messages for the two *dup()* calls. Then, when the client executed the *close()* calls, three `io_close` messages would be generated.

Since the *dup()* functions generate duplicates of the file descriptors, new context information should not be allocated for each one. When the `io_close` messages arrive, because no new context has been allocated for each *dup()*, no *release* of the memory by each `io_close` message should occur either! (If it did, the first close would wipe out the context.)

The resource manager shared library provides default handlers that keep track of the *open()*, *dup()*, and *close()* messages and perform work only for the last close (i.e., the third `io_close` message in the example above).

Multiple thread handling

One of the salient features of the QNX Neutrino RTOS is the ability to use *threads*. By using multiple threads, a resource manager can be structured so that several threads are waiting for messages and then simultaneously handling them.

This thread management is another convenient function provided by the resource manager shared library. Besides keeping track of both the number of threads created and the number of threads waiting, the library also takes care of maintaining the optimal number of threads.

Dispatch functions

The OS provides a set of *dispatch_** functions that:

- allow a common blocking point for managers and clients that need to support multiple message types (e.g., a resource manager could handle its own private message range).
- provide a flexible interface for message types that isn't tied to the resource manager (for clean handling of private messages and pulse codes)
- decouple the blocking and handler code from threads. You can implement the resource manager event loop in your main code. This decoupling also makes for easier debugging, because you can put a breakpoint between the block function and the handler function.

For more information, see the Resource Managers chapter of *Getting Started with QNX Neutrino*, and the *Writing a Resource Manager* guide.

Combine messages

In order to conserve network bandwidth and to provide support for atomic operations, the OS supports *combine messages*. A combine message is constructed by the client's C library and consists of a number of I/O and/or connect messages packaged together into one.

For example, the function *readblock()* allows a thread to atomically perform an *lseek()* and *read()* operation. This is done in the client library by combining the `io_lseek` and `io_read` messages into one. When the resource manager shared library receives the message, it will process both the `io_lseek` and `io_read` messages, effectively making that *readblock()* function behave atomically.

Combine messages are also useful for the *stat()* function. A *stat()* call can be implemented in the client's library as an *open()*, *fstat()*, and *close()*. Instead of generating three separate messages (one for each of the component functions), the library puts them together into one contiguous combine message. This boosts performance, especially over a networked connection, and also simplifies the resource manager, which doesn't need a connect function to handle *stat()*.

The resource manager shared library takes care of the issues associated with breaking out the individual components of the combine message and passing them to the various handler functions supplied. Again, this minimizes the effort associated with writing a resource manager.

Second-level default message handling

Since a large number of the messages received by a resource manager deal with a common set of attributes, the OS provides another level of default handling.

This second level, called the *iofunc_*()* shared library, allows a resource manager to handle functions like *stat()*, *chmod()*, *chown()*, and *lseek() automatically*, without the programmer having to write additional code. As an added benefit, these *iofunc_*()* default handlers implement the POSIX semantics for the messages, again offloading work from the programmer.

Three main structures need to be considered:

- context
- attributes structure
- mount structure

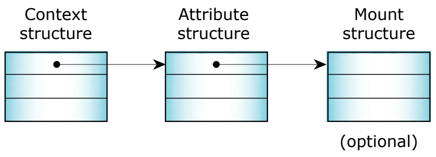


Figure 38: A resource manager is responsible for three data structures.

The first data structure, the context, has already been discussed (see the section on "*Message types*"). It holds data used on a per-open basis, such as the current position into a file (the *lseek()* offset).

Since a resource manager may be responsible for more than one device (e.g., `devc-ser*` may be responsible for **/dev/ser1**, **/dev/ser2**, **/dev/ser3**, etc.), the *attributes* structure holds data on a per-device basis. The attributes structure contains such items as the user and group ID of the owner of the device, the last modification time, etc.

For filesystem (block I/O device) managers, one more structure is used. This is the *mount* structure, which contains data items that are global to the entire mount device.

When a number of client programs have opened various devices on a particular resource, the data structures may look like this:

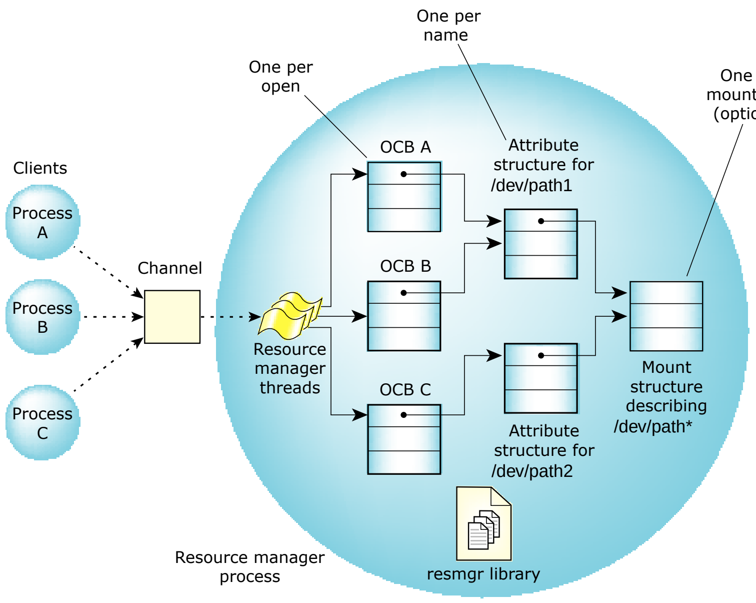


Figure 39: Multiple clients opening various devices.

The *iofunc_*()* default functions operate on the assumption that the programmer has used the default definitions for the context block and the attributes structures. This is a safe assumption for two reasons:

1. The default context and attribute structures contain sufficient information for most applications.
2. If the default structures don't hold enough information, they can be encapsulated within the structures that you've defined.

By definition, the default structures must be the first members of their respective superstructures, allowing clean and simple access to the requisite base members by the *iofunc_*()* default functions:

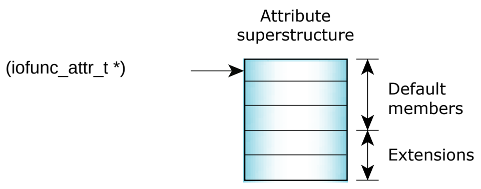


Figure 40: Encapsulating the default data structures used by resource managers.

The library contains *iofunc_*()* default handlers for these client functions:

- *chmod()*
- *chown()*
- *close()*
- *devctl()*
- *fpathconf()*
- *fseek()*
- *fstat()*
- *lock()*
- *lseek()*
- *mmap()*

- *open()*
- *pathconf()*
- *stat()*
- *utime()*

Summary

By supporting pathname space mapping, by having a well-defined interface to resource managers, and by providing a set of libraries for common resource manager functions, the QNX Neutrino RTOS offers the developer unprecedented flexibility and simplicity in developing “drivers” for new hardware—a critical feature for many embedded systems.

For more details on developing a resource manager, see the Resource Managers chapter of *Getting Started with QNX Neutrino*, and the *Writing a Resource Manager* guide.

Chapter 9 Filesystems

The QNX Neutrino RTOS provides a rich variety of filesystems. Like most service-providing processes in the OS, these filesystems execute outside the kernel; applications use them by communicating via messages generated by the shared-library implementation of the POSIX API.

These filesystems are *resource managers* as described in this book. Each filesystem adopts a portion of the pathname space (called a *mountpoint*) and provides filesystem services through the standard POSIX API (*open()*, *close()*, *read()*, *write()*, *lseek()*, etc.). Filesystem resource managers take over a mountpoint and manage the directory structure below it. They also check the individual pathname components for permissions and for access authorizations.

This implementation means that:

- Filesystems may be started and stopped dynamically.
- Multiple filesystems may run concurrently.
- Applications are presented with a single unified pathname space and interface, regardless of the configuration and number of underlying filesystems.
- If you use Qnet, a filesystem running on one node can be made transparently accessible from any other node.

There are some special types of filesystems:

- A *pass-through filesystem* sits in front of another filesystem and manipulates files that are in the underlying filesystem. An example of this is the *Inflator filesystem*, which we'll describe later in this chapter.

  Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

- A *virtual filesystem* is one in which the files or directories aren't necessarily tied directly to the underlying media, perhaps being manufactured on-demand. The **/proc** filesystem is an example; see “Controlling processes via the **/proc** filesystem” in the Processes chapter of the QNX Neutrino *Programmer's Guide*.
- *QNX Trusted Disk* devices provide integrity protection of the underlying disk data when they are used in a secure boot environment.

Filesystems and pathname resolution

You can seamlessly locate and connect to any service or filesystem that's been registered with the process manager. When a filesystem resource manager registers a mountpoint, the process manager creates an entry in the internal mount table for that mountpoint and its corresponding server ID (i.e., the *nd*, *pid*, *chid* identifiers).

This table effectively joins multiple filesystem directories into what users perceive as a single directory. The process manager handles the mountpoint portion of the pathname; the individual filesystem resource managers take care of the remaining parts of the pathname. Filesystems can be registered (i.e., mounted) in any order.

When a pathname is resolved, the process manager contacts all the filesystem resource managers that can handle some component of that path. The result is a collection of file descriptors that can resolve the pathname.

If the pathname represents a directory, the process manager asks all the filesystems that can resolve the pathname for a listing of files in that directory when *readdir()* is called. If the pathname isn't a directory, then the first filesystem that resolves the pathname is accessed.

For more information on pathname resolution, see the section "*Pathname management*" in the chapter on the Process Manager in this guide.

Filesystem classes

The many filesystems available can be categorized into the following classes:

Image

A special filesystem that presents the modules in the image and is always present. Note that the `procnto` process automatically provides an image filesystem and a RAM filesystem.

Block

Traditional filesystems that operate on block devices like hard disks and DVD drives. This includes the *Power-Safe filesystem*, *DOS*, and *Universal Disk Format* filesystems.

Flash

Nonblock-oriented filesystems designed explicitly for the characteristics of flash memory devices. For NOR devices, use the *FFS3* filesystem; for NAND, use *ETFS*.

Network

Filesystems that provide network file access to the filesystems on remote host computers. This includes the *NFS* and *CIFS* (SMB) filesystems.

Pass-through

Filesystems that sit in front of another filesystem and manipulate files that are in the underlying filesystem. This includes the *Inflator filesystem*, which uncompresses files that were previously compressed (using the `deflate` utility).

Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

Virtual

Filesystems in which the files or directories aren't necessarily tied directly to the underlying media, perhaps being manufactured on-demand. This includes the **/proc** filesystem; see “Controlling processes via the **/proc** filesystem” in the Processes chapter of the QNX Neutrino *Programmer's Guide*.

Filesystems as shared libraries

Since it's common to run many filesystems under the QNX Neutrino RTOS, they have been designed as a family of drivers and shared libraries to maximize code reuse. This means the cost of adding an additional filesystem is typically smaller than might otherwise be expected.

Once an initial filesystem is running, the incremental memory cost for additional filesystems is minimal, since only the code to implement the new filesystem protocol would be added to the system.

The various filesystems are layered as follows:

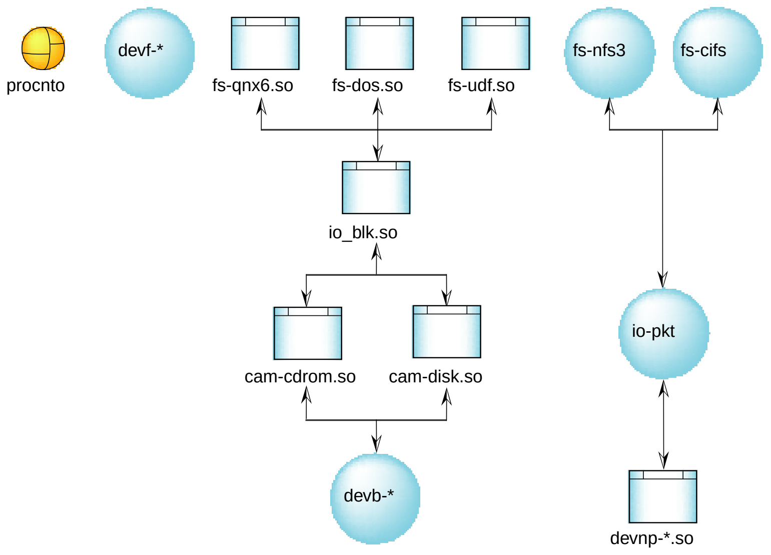


Figure 41: QNX Neutrino filesystem layering.

As shown in this diagram, the filesystems and `io-blk` are implemented as shared libraries (essentially passive blocks of code resident in memory), while the `devb-*` driver is the executing process that calls into the libraries. In operation, the driver process starts first and invokes the block-level shared library (`io-blk.so`). The filesystem shared libraries may be dynamically loaded later to provide filesystem interfaces and services.

A "filesystem" shared library implements a filesystem protocol or "personality" on a set of blocks on a physical disk device. The filesystems aren't built into the OS kernel; rather, they're dynamic entities that can be loaded or unloaded on demand.

For example, a removable storage device (removable cartridge disk, etc.) may be inserted at any time, with any of a number of filesystems stored on it. While the hardware the driver interfaces to is unlikely to change dynamically, the on-disk data structure could vary widely. The dynamic nature of the filesystem copes with this very naturally.

io-blk

Most of the filesystem shared libraries ride on top of the Block I/O module.

The `io-blk.so` module also acts as a resource manager and exports a block-special file for each physical device. For a system with two hard disks the default files would be:

/dev/hd0

First hard disk.

/dev/hd1

Second hard disk.

These files represent each raw disk and may be accessed using all the normal POSIX file primitives (*open()*, *close()*, *read()*, *write()*, *lseek()*, etc.). Although the `io-blk` module can support a 64-bit offset on seek, the driver interface is 32-bit, allowing access to 2-terabyte disks.

Builtin RAM disk

The `io-blk` module supports an internal RAM-disk device that can be created via a command-line option (`blk ramdisk=`*size*).

Since this RAM disk is internal to the `io-blk` module (rather than created and maintained by an additional device driver such as `devb-ram`), performance is significantly better than that of a dedicated RAM-disk driver.

By incorporating the RAM-disk device directly at the `io-blk` layer, the device's data memory parallels the main cache, so I/O operations to this device can bypass the buffer cache, eliminating a memory copy yet still retaining coherency. Contrast this with a driver-level implementation (e.g., `devb-ram`) where transparently presenting the RAM as a block device involves additional memory copies and duplicates data in the buffer cache. Inter-DLL callouts are also eliminated. In addition, there are benefits in terms of installation footprint for systems that have a hard disk and also want a RAM disk: only the single driver is needed.

Partitions

The QNX Neutrino RTOS complies with the de facto industry standard for partitioning a disk.

This allows a number of filesystems to share the same physical disk. Each partition is also represented as a block-special file, with the partition type appended to the filename of the disk it's located on. In the above “two-disk” example, if the first disk had a Power-Safe partition and a DOS partition, while the second disk had only a Power-Safe partition, then the default files would be:

/dev/hd0

First hard disk

/dev/hd0t6

DOS partition on first hard disk

/dev/hd0t179

Power-Safe partition on first hard disk

/dev/hd1

Second hard disk

/dev/hd1t179

Power-Safe partition on second hard disk

The following list shows some typical assigned partition types:

Type	Filesystem
1	DOS (12-bit FAT)
4	DOS (16-bit FAT; partitions <32M)
5	DOS Extended Partition (enumerated but not presented)

Type	Filesystem
6	DOS 4.0 (16-bit FAT; partitions 32M)
7	OS/2 HPFS
7	Windows NT
11	DOS 32-bit FAT; partitions up to 2047G
12	Same as Type 11, but uses Logical Block Address `Int 13h` extensions
14	Same as Type 6, but uses Logical Block Address `Int 13h` extensions
15	Same as Type 5, but uses Logical Block Address `Int 13h` extensions
77	QNX 4
78	QNX 4
79	QNX 4
99	UNIX
131	Linux (Ext2)
175	Apple Macintosh HFS or HFS Plus
177	QNX Power-Safe POSIX partition (secondary)
178	QNX Power-Safe POSIX partition (secondary)
179	QNX Power-Safe POSIX partition

Buffer cache

The `io-blk` shared library implements a *buffer cache* that all filesystems inherit. The buffer cache attempts to store frequently accessed filesystem blocks in order to minimize the number of times a system has to perform a physical I/O to the disk.

Read operations are synchronous; write operations are usually asynchronous. When an application writes to a file, the data enters the cache, and the filesystem manager immediately replies to the client process to indicate that the data has been written. The data is then written to the disk.

Critical filesystem blocks such as bitmap blocks, directory blocks, extent blocks, and inode blocks are written immediately and synchronously to disk.

Applications can modify write behavior on a file-by-file basis. For example, a database application can cause all writes for a given file to be performed synchronously. This would ensure a high level of file integrity in the face of potential hardware or power problems that might otherwise leave a database in an inconsistent state.

Filesystem limitations

POSIX defines the set of services a filesystem must provide. However, not all filesystems are capable of delivering all those services.

Filesystem	Access date	Modification date	Status change date	Filename length[a]	Permissions	Directories	Hard links	Soft links	Decompression on read
Image	No	No	No	255	Yes	No	No	No	No
RAM[b]	Yes	Yes	Yes	255	Yes	No	No	No	No
ETFS	Yes	Yes	Yes	91	Yes	Yes	No	Yes	No
Power-Safe	Yes	Yes	Yes	510	Yes	Yes	Yes	Yes	No
DOS	Yes[c]	Yes	No	8.3[d]	No	Yes	No	No	No
NTFS	Yes	Yes	Yes	755	No	Yes	No	No	Yes
UDF	Yes	Yes	Yes	254	Yes	Yes	No	No	No
HFS, HFS Plus	Yes	Yes	Yes	255[e]	Yes	Yes	Yes	Yes	No
FFS3	No	Yes	Yes	255	Yes	Yes	No	Yes	No
NFS	Yes	Yes	Yes	—[f]	Yes[f]	Yes	Yes[f]	Yes[f]	No
CIFS	No	Yes	No	—[f]	Yes[f]	Yes	No	No	No
Ext2	Yes	Yes	Yes	255	Yes	Yes	Yes	Yes	No

[a] Our internal representation for file names is UTF-8, which uses a variable number of bytes per character. Many on-disk formats instead use UCS2, which is a fixed number (2 bytes). Thus a length limit in characters may be 1, 2, or 3 times that number in bytes, as we convert from on-disk to OS representation. The lengths for the Power-Safe and EXT2 filesystems are in bytes; those for UDF and DOS/VFAT are in characters.

[b] The RAM “filesystem” (**/dev/shmem**) isn't really a filesystem; it's a window onto the shared memory names that has *some* filesystem-like characteristics. See “*Builtin RAM disk*” later in this chapter.

[c] VFAT or FAT32 (Windows 95 or later).

[d] 255-character filename lengths used by VFAT or FAT32 (e.g., Windows 95).

[e] 31 on HFS.

[f] Limited by the remote filesystem.

Image filesystem

Every QNX Neutrino system image provides a simple *read-only* filesystem that presents the set of files built into the OS image.

Since this image may include both executables and data files, this filesystem is sufficient for many embedded systems. If additional filesystems are required, they would be placed as modules within the image where they can be started as needed.

RAM “filesystem”

Every QNX Neutrino system also provides a simple RAM-based “filesystem” that allows read/write files to be placed under **/dev/shmem**.

Note that **/dev/shmem** isn't actually a filesystem. It's a window onto the shared memory names that happens to have *some* filesystem-like characteristics.

This RAM filesystem finds the most use in tiny embedded systems where persistent storage across reboots isn't required, yet where a small, fast, *temporary-storage* filesystem with limited features is called for.

The filesystem comes for free with `procnto` and doesn't require any setup. You can simply create files under **/dev/shmem** and grow them to any size (depending on RAM resources).

Although the RAM filesystem itself doesn't support hard or soft links or directories, you can create a link to it by using process-manager links. For example, you could create a link to a RAM-based **/tmp** directory:

```
ln -sP /dev/shmem /tmp
```

This tells `procnto` to create a process manager link to **/dev/shmem** known as “**/tmp**.” Application programs can then open files under **/tmp** as if it were a normal filesystem.

In order to minimize the size of the RAM filesystem code inside the process manager, this filesystem specifically doesn't include “big filesystem” features such as file locking and directory creation.

Embedded transaction filesystem (ETFS)

ETFS implements a high-reliability filesystem for use with embedded solid-state memory devices, particularly NAND flash memory.

The filesystem supports a fully hierarchical directory structure with POSIX semantics as shown in the table above.

ETFS is a filesystem composed entirely of *transactions*. Every write operation, whether of user data or filesystem metadata, consists of a transaction. A transaction either succeeds or is treated as if it never occurred.

Transactions never overwrite live data. A write in the middle of a file or a directory update always writes to a new unused area. In this way, if the operation fails part way through (due to a crash or power failure), the old data is still intact.

Some log-based filesystems also operate under the principle that live data is never overwritten. But ETFS takes this to the extreme by turning everything into a log of transactions. The filesystem hierarchy is built on the fly by processing the log of transactions in the device. This scan occurs at startup, but is designed such that only a small subset of the data is read and CRC-checked, resulting in faster startup times without sacrificing reliability.

Transactions are position-independent in the device and may occur in any order. You could read the transactions from one device and write them in a different order to another device. This is important because it allows bulk programming of devices containing bad blocks that may be at arbitrary locations.

This design is well-suited for NAND flash memory. NAND flash is shipped with factory-marked bad blocks that may occur in any location.

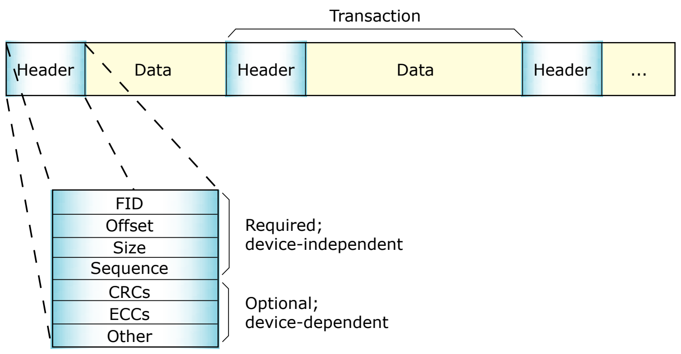


Figure 42: ETFS is a filesystem composed entirely of transactions.

Inside a transaction

Each transaction consists of a header followed by data. The header contains the following:

FID

A unique file ID that identifies which file the transaction belongs to.

Offset

The offset of the data portion within the file.

Size

The size of the data portion.

Sequence

A monotonically increasing number (to enable time ordering).

CRCs

Data integrity checks (for NAND, NOR, SRAM).

ECCs

Error correction (for NAND).

Other

Reserved for future expansion.

Types of storage media

Although best for NAND devices, ETFS also supports other types of embedded storage media by using driver classes as follows:

Class	CRC	ECC	Wear-leveling erase	Wear-leveling read	Cluster size
NAND 512+16	Yes	Yes	Yes	Yes	1 KB
NAND 2048+64	Yes	Yes	Yes	Yes	2 KB
RAM	No	No	No	No	1 KB
SRAM	Yes	No	No	No	1 KB
NOR	Yes	No	Yes	No	1 KB

Although ETFS can support NOR flash, we recommend instead the *FFS3* filesystem (`devf-*`), which is designed explicitly for NOR flash devices.

Reliability features

ETFS is designed to survive across a power failure, even during an active flash write or block erase. The following features contribute to its reliability:

- dynamic wear-leveling
- static wear-leveling
- CRC error detection
- ECC error correction
- read degradation monitoring with automatic refresh
- transaction rollback

- atomic file operations
- automatic file defragmentation.

Dynamic wear-leveling

Flash memory allows a limited number of erase cycles on a flash block before the block will fail. This number can be as low as 100,000. ETFS tracks the number of erases on each block. When selecting a block to use, ETFS attempts to spread the erase cycles evenly over the device, dramatically increasing its life. The difference can be extreme: from usage scenarios of failure within a few days without wear-leveling to over 40 years with wear-leveling.

Static wear-leveling

Filesystems often consist of a large number of static files that are read but not written. These files will occupy flash blocks that have no reason to be erased. If the majority of the files in flash are static, this will cause the remaining blocks containing dynamic data to wear at a dramatically increased rate.

ETFS notices these underworked static blocks and forces them into service by copying their data to an overworked block. This solves two problems: it gives the overworked block a rest, since it now contains static data, and it forces the underworked static block into the dynamic pool of blocks.

CRC error detection

Each transaction is protected by a cyclic redundancy check (CRC). This ensures quick detection of corrupted data, and forms the basis for the rollback operation of damaged or incomplete transactions at startup. The CRC can detect multiple bit errors that may occur during a power failure.

ECC error correction

On a CRC error, ETFS can apply error correction coding (ECC) to attempt to recover the data. This is suitable for NAND flash memory, in which single-bit errors may occur during normal usage. An ECC error is a warning signal that the flash block the error occurred in may be getting weak, i.e., losing charge.

ETFS marks the weak block for a *refresh* operation, which copies the data to a new flash block and erases the weak block. The erasure recharges the flash block.

Read degradation monitoring with automatic refresh

Each read operation within a NAND flash block weakens the charge maintaining the data bits. Most devices support about 100,000 reads before there's danger of losing a bit. The ECC recovers a single-bit error, but may not be able to recover multi-bit errors.

ETFS solves this by tracking reads and marking blocks for refresh before the 100,000 read limit is reached.

Transaction rollback

When ETFS starts, it processes all transactions and rolls back (discards) the last partial or damaged transaction. The rollback code is designed to handle a power failure during a rollback operation, thus allowing the system to recover from multiple nested faults. The validity of a transaction is protected by CRC codes on each transaction.

Atomic file operations

ETFS implements a very simple directory structure on the device, allowing significant modifications with a single flash write. For example, the move of a file or directory to another directory is often a multistage operation in most filesystems. In ETFS, a move is accomplished with a single flash write.

Automatic file defragmentation

Log-based filesystems often suffer from fragmentation, since each update or write to an existing file causes a new transaction to be created. ETFS uses write-buffering to combine small writes into larger write transactions in an attempt to minimize fragmentation caused by lots of very small transactions. ETFS also monitors the fragmentation level of each file and will do a background defragmenting operation on files that do become badly fragmented. Note that this background activity will always be preempted by a user data request in order to ensure immediate access to the file being defragmented.

Power-Safe filesystem

The Power-Safe filesystem is a reliable disk filesystem that can withstand power failures without losing or corrupting data. It was designed for and is intended for traditional rotating hard disk drive media.

This filesystem is supported by the **fs-qnx6.so** shared object.

Problems with existing disk filesystems

Although existing disk filesystems are designed to be robust and reliable, there's still the possibility of losing data, depending on what the filesystem is doing when a catastrophic failure (such as a power failure) occurs.

For example:

- Each sector on a hard disk includes a 4-byte error-correcting code (ECC) that the drive uses to catch hardware errors and so on. If the driver is writing the disk when the power fails, then the heads are removed to prevent them from crashing on the surface, leaving the sector half-written with the new content. The next time you try to read that block—or sector—the inconsistent ECC causes the read to fail, so you lose both the old and new content.

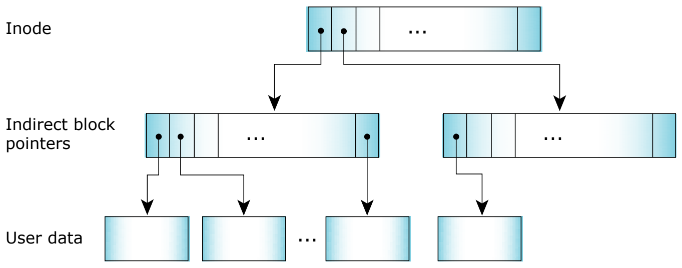
  You can get hard drives that offer atomic sector upgrades and promise you that either all of the old or new data in the sector will be readable, but these drives are rare and expensive.
- Some filesystem operations require updating multiple on-disk data structures. For example, if a program calls *unlink()*, the filesystem has to update a bitmap block, a directory block, and an inode, which means it has to write three separate blocks. If the power fails between writing these blocks, the filesystem will be in an inconsistent state on the disk. Critical filesystem data, such as updates to directories, inodes, extent blocks, and the bitmap are written synchronously to the disk in a carefully chosen order to reduce—but not eliminate—this risk.
- If the root directory, the bitmap, or inode file (all in the first few blocks of the disk) gets corrupted, you wouldn't be able to mount the filesystem at all. You might be able to manually repair the system, but you need to be very familiar with the details of the filesystem structure.

Copy-on-write filesystem

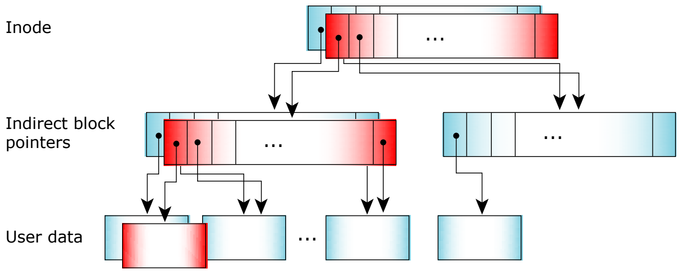
To address the problems associated with existing disk filesystems, the Power-Safe filesystem never overwrites live data; it does all updates using copy-on-write (COW), assembling a new view of the filesystem in unused blocks on the disk. The new view of the filesystem becomes “live” only when all the updates are safely written on the disk. *Everything* is COW: both metadata and user data are protected.

To see how this works, let's consider how the data is stored. A Power-Safe filesystem is divided into logical blocks, the size of which you can specify when you use `mkqnx6fs` to format the filesystem. Each inode includes 16 pointers to blocks. If the file is smaller than 16 blocks, the inode points to the data blocks directly. If the file is any bigger, those 16 blocks become pointers to more blocks, and so on.

The final block pointers to the real data are all in the leaves and are all at the same level. In some other filesystems—such as EXT2—a file always has some direct blocks, some indirect ones, and some double indirect, so you go to different levels to get to different parts of the file. With the Power-Safe filesystem, all the user data for a file is at the same level.



If you change some data, it's written in one or more unused blocks, and the original data remains unchanged. The list of indirect block pointers must be modified to refer to the newly used blocks, but again the filesystem copies the existing block of pointers and modifies the copy. The filesystem then updates the inode—once again by modifying a copy—to refer to the new block of indirect pointers. When the operation is complete, the original data and the pointers to it remain intact, but there's a new set of blocks, indirect pointers, and inode for the modified data:



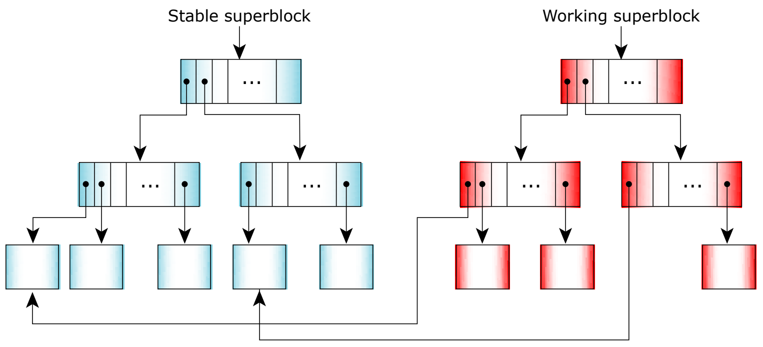
This has several implications for the COW filesystem:

- The bitmap and inodes are treated in the same way as user files.
- Any filesystem block can be relocated, so there aren't any fixed locations.
- The filesystem must be completely self-referential.

A *superblock* is a global root block that contains the inodes for the system bitmap and inodes files. A Power-Safe filesystem maintains *two* superblocks:

- a stable superblock that reflects the original version of all the blocks
- a working superblock that reflects the modified data

The working superblock can include pointers to blocks in the stable superblock. These blocks contain data that hasn't yet been modified. The inodes and bitmap for the working superblock grow from it.



A *snapshot* is a consistent view of the filesystem (simply a committed superblock). To take a snapshot, the filesystem:

1. Locks the filesystem to make sure that it's in a stable state; all client activity is suspended, and there must be no active operations.
2. Writes all the copied blocks to disk. The order isn't important, so it can be optimized.
3. Forces the data to be synchronized to disk, including flushing any hardware track cache.
4. Constructs the superblock, recording the new location of the bitmap and inodes, incrementing its sequence number, and calculating a CRC.
5. Writes the superblock to disk.
6. Switches between the working and committed views. The old versions of the copied blocks are freed and become available for use.

To mount the disk at startup, the filesystem simply reads the superblocks from disk, validates their CRCs, and then chooses the one with the higher sequence number. There's no need to replay a transaction log. The time it takes to mount the filesystem is the time it takes to read a couple of blocks.

If the drive doesn't support synchronizing, **fs-qnx6.so** can't guarantee that the filesystem is power-safe. Before using this filesystem on devices—such as USB/Flash devices—other than traditional rotating hard disk drive media, check to make sure that your device meets the filesystem's requirements. For more information, see “Required properties of the device” in the entry for **fs-qnx6.so** in the *Utilities Reference*.

Performance

The Copy on Write (COW) method has some drawbacks:

- Each change to user data can cause up to a dozen blocks to be copied and modified, because the filesystem never modifies the inode and indirect block pointers in place; it has to copy the blocks to a new location and modify the copies. Thus, write operations are longer.
- When taking a snapshot, the filesystem must force all blocks fully to disk before it commits the superblock.

However:

- There's no constraint on the order in which the blocks (aside from the superblock) can be written.
- The new blocks can be allocated from any free, contiguous space.

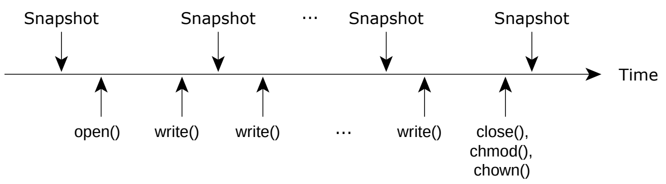
The performance of the filesystem depends on how much buffer cache is available, and on the frequency of the snapshots. Snapshots occur periodically (every 10 seconds, or as specified by the `snapshot` option to **fs-qnx6.so**), and also when you call *sync()* for the entire filesystem, or *fsync()* for a single file.

Synchronization is at the filesystem level, not at that of individual files, so *fsync()* is potentially an expensive operation; the Power-Safe filesystem ignores the O_SYNC flag.

You can also turn snapshots off if you're doing some long operation, and the intermediate states aren't useful to you. For example, suppose you're copying a very large file into a Power-Safe filesystem. The `cp` utility is really just a sequence of basic operations:

- an `open(O_CREAT|O_TRUNC)` to make the file
- a bunch of *write()* operations to copy the data
- a *close()*, *chmod()*, and *chown()* to copy the metadata

If the file is big enough so that copying it spans snapshots, you have on-disk views that include the file not existing, the file existing at a variety of sizes, and finally the complete file copied and its IDs and permissions set:



Each snapshot is a valid point-in-time view of the filesystem (i.e., if you've copied 50 MB, the size is 50 MB, and all data up to 50 MB is also correctly copied and available). If there's a power failure, the filesystem is restored to the most recent snapshot. But the filesystem has no concept that the sequence of *open()*, *write()*, and *close()* operations is really one higher-level operation, `cp`. If you want the higher-level semantics, disable the snapshots around the `cp`, and then the middle snapshots won't happen, and if a power failure occurs, the file will either be complete, or not there at all.

For information about using this filesystem, see "Power-Safe filesystem" in the Working with Filesystems chapter of the QNX Neutrino *User's Guide*.

Encryption

You can encrypt all or parts of a Power-Safe filesystem in order to protect its contents.

- In order to use filesystem encryption, download the Encrypted Filesystem package from the QNX Software Center.
- When you use an encrypted filesystem, read/write throughput rates and performance are lower and CPU usage is higher than when you use a nonencrypted filesystem.

You might need to encrypt different parts of the filesystem with different keys, and you might not need to encrypt some parts, so the Power-Safe filesystem lets you create multiple *encryption domains*, which you can lock or unlock as needed.

A domain can contain any number of files or directories, but a file or directory can belong to at most one domain. If you assign a domain to a directory, all files subsequently created in that directory are encrypted and inherit that domain. Assigning a domain to a directory doesn't introduce encryption to any files or directories that are already in it.

You can simply assign a domain to a directory or an empty file; the filesystem treats them in the same way because they don't have any content to encrypt. If you want to encrypt a file that isn't empty, you must make it *migrate* to a domain because the filesystem needs to decrypt the file, assign the new domain to it, and then encrypt the file again.

During operation, files that are assigned to a domain are encrypted, and the files' contents are available only when the associated domain is unlocked. When a domain is unlocked, all the files and directories under that domain—regardless of their locations in the volume—are unlocked as well, and are therefore accessible (as per basic file permissions). When a domain is locked, any access to file content belonging to that domain is denied.

Locking and unlocking operations apply to an entire domain, not to specific files or directories.

Domain 0 (FS_CRYPTO_UNASSIGNED_DOMAIN) is always unlocked, and its contents are unencrypted; you can design your system to use any other domains. Valid domain numbers are 0–119.

In order to use encryption, you must set the `crypto=` option for **fs-qnx6.so**. You can then use `fsencrypt` to manage the encryption. The `chkqnx6fs` utility automatically identifies the encryption format and verifies the integrity of the encryption data. It's also possible for you to use `fsencrypt` to enable encryption after you've formatted the volume and added content, but you must have set the `crypto=` option when you started **fs-qnx6.so**.

Key types

Key type	Description
File key	Private and randomly generated at the time the file is created (if the file is assigned to a domain). Holds some information about the file to ensure integrity between a file and its key. Used to encrypt file data, and is encrypted by a domain key. Keys are managed by the filesystem and are hidden from the user.
Domain key	Private and randomly generated at the time the domain is created. Used to encrypt all the file keys that belong to its domain, and is encrypted by a domain master key. Keys are managed by the filesystem and are hidden from the user.
Master key	Optionally public, as it is supplied and managed by a third party (not the filesystem). Used to encrypt the domain key, required on domain creation and subsequent unlock requests.

Encryption types

The Power-Safe filesystem supports the following types of encryption:

Domain-encryption type	Constant	Description	Key length
0	FS_CRYPTO_TYPE_NONE	No encryption	—
1	FS_CRYPTO_TYPE_XTS	AES-256, in XTS mode. The two keys are randomly generated.	512 bits
2	FS_CRYPTO_TYPE_CBC	AES-256, in CBC mode	256 bits

Domain-encryption type	Constant	Description	Key length
3–99	—	Reserved for future use	—

Interface usage

To manage encryption from the command line, use `fsencrypt`; its `-c` option lets you specify the command to run. From your code, use the **fscrypto** library. You need to include both the **<fs_crypto_api.h>** and **<sys/fs_crypto.h>** header files. Many of the APIs return EOK on success and have a reply argument that provides more information.

API	`fsencrypt` command	Description
fs_crypto_check()	`check`	Determine if the underlying filesystem supports encryption
fs_crypto_domain_add()	`create`	Create the given domain/type if it doesn't already exist. You need to provide a 64-bit encryption key. From a program, you can create a locked or unlocked domain; `fsencrypt` always creates an unlocked domain.
fs_crypto_domain_key_change()	`change-key`	Change the master domain key used to encrypt the domain key
fs_crypto_domain_key_check()	`check-key`	Determine if a given domain key is valid
fs_crypto_domain_key_size()	—	Return the size of keys needed for filesystem encryption
fs_crypto_domain_lock()	`lock`	Lock a domain, preventing access to the original contents of any file belonging to the given domain
fs_crypto_domain_query()	`query`, `query-all`	Get status information for a domain
fs_crypto_domain_remove()	`destroy`	Destroy a domain. You must be in the group that owns the filesystem's mountpoint. It isn't possible to retrieve any files in the domain after it's been destroyed.
fs_crypto_domain_unlock()	`unlock`	Unlock a domain, given the appropriate key data
fs_crypto_enable(), *fs_crypto_enable_option()*	`enable`	Enable encryption support on a volume that wasn't set up for it at formatting time
fs_crypto_file_get_domain()	`get`	Return the domain of a file or directory, if assigned
fs_crypto_file_set_domain()	`set`	Assign a given domain to the path (a regular file or a directory). Regular files must have a length of zero. The domain replaces any domain previously assigned to the path.

API	`fsencrypt` command	Description
fs_crypto_key_gen()	`-K` or `-k` option	Generate an encryption key from a password
fs_crypto_set_logging()	`-l` and `-v` options	Set the logging destination and verbosity

The library also includes some functions that you can use to move existing files and directories into an encryption domain. To do this, you unlock the source and destination domains, tag the files and directories that you want to move, and then start the migration, which the filesystem does in the background. These functions include:

API	`fsencrypt` command	Description
fs_crypto_migrate_control()	`migrate-start`, `migrate-stop`, `migrate-delay`, `migrate-units`	Control encryption migration within the filesystem
fs_crypto_migrate_path()	`migrate-path`	Mark an entire directory for migration into an encryption domain
fs_crypto_migrate_status()	`migrate-status`	Get the status of migration in the filesystem
fs_crypto_migrate_tag()	`migrate-tag`, `tag`	Mark a file for migration into an encryption domain

Examples

Here are some examples of the way you can use `fsencrypt` to manage filesystem encryption:

- Determine if encryption is supported or enabled:

  ```
  $ fsencrypt -vc check -p /
  ENCRYPTION_CHECK(Path:'/') FAILED: (18) - 'No support'
  ```

- Enable encryption on an existing filesystem:

  ```
  $ fsencrypt -vc enable -p /
  ENCRYPTION_CHECK(Path:'/') SUCCESS
  $ fsencrypt -vc check -p /
  ENCRYPTION_CHECK(Path:'/') NOTICE: Encryption is SUPPORTED)
  ```

  This change is irreversible and forces two consecutive disk transactions to rewrite some data in the superblocks.

- Determine if a file is encrypted. Files with the domain number of 0 aren't encrypted. A nonzero value means the file is assigned to a domain. Access to the file's original contents is determined by the status of the domain. The example below shows that the named file is assigned to domain 10:

  ```
  $ fsencrypt -vcget -p /accounts/1000/secure/testfile
  GET_DOMAIN(Path:'/accounts/1000/secure/testfile') = 10 SUCCESS
  ```

- Determine if a domain is locked. If a domain is locked, reading the files within that domain yields encrypted data; unlocked domains yield the original file contents. “Unused” domains are ones that haven't yet been created. In the example below, domain 11 hasn't yet been created, and domain 10 is currently unlocked:

  ```
  $ fsencrypt -vc query -p/ -d11
  QUERY_DOMAIN(Path:'/', Domain:11) NOTICE: Domain is UNUSED
  $ fsencrypt -vc query -p/ -d10
  QUERY_DOMAIN(Path:'/', Domain:10) NOTICE: Domain is UNLOCKED
  ```

DOS Filesystem

The DOS Filesystem, **fs-dos.so**, provides transparent access to DOS disks, so you can treat DOS filesystems as though they were POSIX filesystems. This transparency allows processes to operate on DOS files without any special knowledge or work on their part.

The structure of the DOS filesystem on disk is old and inefficient, and lacks many desirable features. Its only major virtue is its portability to DOS and Windows environments. You should choose this filesystem only if you need to transport DOS files to other machines that require it. Consider using the Power-Safe filesystem alone if DOS file portability isn't an issue or in conjunction with the DOS filesystem if it is.

If there's no DOS equivalent to a POSIX feature, **fs-dos.so**, with either return an error or a reasonable default. For example, an attempt to create a *link()* will result in the appropriate *errno* being returned. On the other hand, if there's an attempt to read the POSIX times on a file, **fs-dos.so** will treat any of the *unsupported* times the same as the last write time.

DOS version support

The **fs-dos.so** program supports hard disk partitions from DOS version 2.1 to Windows 98 with long filenames.

DOS text files

DOS terminates each line in a text file with two characters (CR/LF), while POSIX (and most other) systems terminate each line with a single character (LF). Note that **fs-dos.so** makes no attempt to translate text files being read. Most utilities and programs aren't affected by this difference.

Note also that some very old DOS programs may use a **Ctrl–Z** (`^Z`) as a file terminator. This character is also passed through without modification.

QNX-to-DOS filename mapping

In DOS, a filename can't contain any of the following characters:

```
/ \ [ ] : * | + = ; , ?
```

An attempt to create a file that contains one of these invalid characters will return an error. DOS (8.3 format) also expects all alphabetical characters to be uppercase, so **fs-dos.so** maps these characters to uppercase when creating a filename on disk. But it maps a filename to lowercase by default when returning a filename to a QNX Neutrino application, so that QNX Neutrino users and programs can always see and type lowercase (via the `sfn=`*sfn_mode* option).

Handling filenames

You can specify how you want **fs-dos.so** to handle long filenames (via the `lfn=`*lfn_mode* option):

- Ignore them—display/create only 8.3 filenames.
- Show them—if filenames are longer than 8.3 or if mixed case is used.

- Always create both short and long filenames.

If you use the `ignore` option, you can specify whether or not to silently truncate filename characters beyond the 8.3 limit.

International filenames

The DOS filesystem supports DOS “code pages” (international character sets) for locale filenames. Short 8.3 names are stored using a particular character set (typically the most common extended characters for a locale are encoded in the 8th-bit character range). All the common American as well as Western and Eastern European code pages (437, 850, 852, 866, 1250, 1251, 1252) are supported. If you produce software that must access a variety of DOS/Windows hard disks, or operate in non-US-English countries, this feature offers important portability—filenames will be created with both a Unicode and locale name and are accessible via either name.

The DOS filesystem supports international text in filenames only. No attempt is made to be aware of data contents, with the sole exception of Windows “shortcut” (**.LNK**) files, which will be parsed and translated into symbolic links if you've specified that option (`lnk=`*lnk_mode*).

DOS volume labels

DOS uses the concept of a volume label, which is an actual directory entry in the root of the DOS filesystem. To distinguish between the volume label and an actual DOS directory, **fs-dos.so** reports the volume label according to the way you specify its `vollabel` option. You can choose to:

- Ignore the volume label.
- Display the volume label as a name-special file.
- Display the volume label as a name-special file with an equal sign (=) as the first character of the volume name (the default).

DOS-QNX permission mapping

DOS doesn't support all the permission bits specified by POSIX. It has a READ_ONLY bit in place of separate READ and WRITE bits; it doesn't have an EXECUTE bit. When a DOS file is created, the DOS READ_ONLY bit is set if all the POSIX WRITE bits are off. When a DOS file is accessed, the POSIX READ bit is always assumed to be set for user, group, and other.

Since you can't execute a file that doesn't have EXECUTE permission, **fs-dos.so** has an option (`exe=`*exec_mode*) that lets you specify how to handle the POSIX EXECUTE bit for executables.

File ownership

Although the DOS file structure doesn't support user IDs and group IDs, **fs-dos.so** (by default) doesn't return an error code if an attempt is made to change them. An error isn't returned because a number of utilities attempt to do this and failure would result in unexpected errors. The approach taken is “you can change anything to anything since it isn't written to disk anyway.”

The `posix=` options let you set stricter POSIX checks and enable POSIX emulation. For example, in POSIX mode, an error of EINVAL is flagged for attempts to do any of the following:

- Set the user ID or group ID to something other than the default (**root**).
- Remove an `r` (read) permission.
- Set an `s` (set ID on execution) permission.

If you set the `posix` option to `emulate` (the default) or `strict`, you get the following benefits:

- The **.** and **..** directory entries are created in the **root** directory.
- The directory size is calculated.
- The number of links in a directory is calculated, based on its subdirectories.

FFS3 filesystem

The FFS3 filesystem drivers implement a POSIX-like filesystem on NOR flash memory devices. The drivers are standalone executables that contain both the flash filesystem code and the flash device code. There are versions of the FFS3 filesystem driver for different embedded systems hardware as well as PCMCIA memory cards.

The naming convention for the drivers is `devf-`*system*, where *system* describes the embedded system.

To find out what flash devices we currently support, refer to the following sources:

- the **boards** and **mtd-flash** directories under *bsp_working_dir***/src/hardware/flash**
- QNX Neutrino RTOS docs (`devf-*` entries in the *Utilities Reference*)
- the QNX Software Systems website (*www.qnx.com*)

Along with the prebuilt flash filesystem drivers, including the "generic" driver (`devf-generic`), we provide the libraries and source code that you'll need to build custom flash filesystem drivers for different embedded systems.

Organization

The FFS3 filesystem drivers support one or more logical flash drives. Each logical drive is called a *socket*, which consists of a contiguous and homogeneous region of flash memory. For example, in a system containing two different types of flash device at different addresses, where one flash device is used for the boot image and the other for the flash filesystem, each flash device would appear in a different socket.

Each socket may be divided into one or more partitions. Two types of partitions are supported: raw partitions and flash filesystem partitions.

Raw partitions

A raw partition in the socket is any partition that doesn't contain a flash filesystem. The driver doesn't recognize any filesystem types other than the flash filesystem. A raw partition may contain an image filesystem or some application-specific data.

The filesystem will make accessible through a raw mountpoint (see below) any partitions on the flash that aren't flash filesystem partitions. Note that the flash filesystem partitions are available as raw partitions as well.

Filesystem partitions

A flash filesystem partition contains the POSIX-like flash filesystem, which uses a QNX Software Systems proprietary format to store the filesystem data on the flash devices. This format isn't compatible with either the Microsoft FFS2 or PCMCIA FTL specification.

The filesystem allows files and directories to be freely created and deleted. It recovers space from deleted files using a reclaim mechanism similar to garbage collection.

When you start the flash filesystem driver, it will by default mount any partitions it finds in the socket. There can only be one instance of the driver per flash device, and the driver must be provided with the full size of the flash chip. Note that you can specify the mountpoint using `mkefs` or `flashctl` (e.g., **/flash**).

Mountpoint	Description
/dev/fs*X*	Raw mountpoint socket *X*
/dev/fs*X***p***Y*	Raw mountpoint socket *X* partition *Y*
/fs*X***p***Y*	Filesystem mountpoint socket *X* partition *Y*
/fs*X***p***Y***/.cmp**	Filesystem compressed mountpoint socket *X* partition *Y*

Features

The FFS3 filesystem supports many advanced features, such as POSIX compatibility, multiple threads, background reclaim, fault recovery, transparent decompression, endian-awareness, wear-leveling, and error-handling.

POSIX

The filesystem supports the standard POSIX functionality (including long filenames, access privileges, random writes, truncation, and symbolic links) with the following exceptions:

- You can't create hard links.
- Access times aren't supported (but file modification times and attribute change times are).

These design compromises allow this filesystem to remain small and simple, yet include most features normally found with block device filesystems.

Storage efficiency

The filesystem uses a fixed-size 32-byte extent header for each piece of user data stored on the filesystem. The filesystem has a 512-byte “append buffer” that's used to coalesce small appends to a file before flushing them out to disk. There's no other caching of user data in the filesystem (read or write). The maximum extent size is currently limited to 4096 bytes of data (plus the 32-byte header) for files which are created at runtime, although `mkefs` can create 16 KB extents.

Wear leveling

Wear leveling is performed on blocks within a given partition. If the flash is split into multiple partitions, then each partition has a smaller pool to level across. Blocks never migrate from one partition to another.

The most common type of reclaim in the filesystem (a foreground reclaim) always does a double-reclaim operation. The first reclaim always takes the block in the partition with the lowest erase count, and swaps it with the spare. Then the block that requires the reclaim is swapped with the spare. This has shown to result in near-perfect wear leveling when foreground reclaims are used.

Background reclaim

The FFS3 filesystem stores files and directories as a linked list of extents, which are marked for deletion as they're deleted or updated. Blocks to be reclaimed are chosen using a simple algorithm that finds the block with the most space to be reclaimed, while keeping level the amount of wear of each individual block. This wear-leveling increases the MTBF (mean time between failures) of the flash devices, thus increasing their longevity.

The background reclaim process is performed when there isn't enough free space. The reclaim process first copies the contents of the reclaim block to an empty spare block, which then replaces the reclaim block. The reclaim block is then erased. Unlike rotating media with a mechanical head, proximity of data isn't a factor with a flash filesystem, so data can be scattered on the media without loss of performance.

Fault recovery

The filesystem has been designed to minimize corruption due to accidental loss-of-power faults. Updates to extent headers and erase block headers are always executed in carefully scheduled sequences. These sequences allow the recovery of the filesystem's integrity in the case of data corruption.

Note that properly designed flash hardware is essential for effective fault-recovery systems. In particular, special reset circuitry must be in place to hold the system in “reset” before power levels drop below critical. Otherwise, spurious or random bus activity can form write/erase commands and corrupt the flash beyond recovery.

Rename operations are guaranteed atomic, even through loss-of-power faults. This means, for example, that if you lost power while giving an image or executable a new name, you would still be able to access the file via its old name upon recovery.

When the FFS3 filesystem driver is started, it scans the state of every extent header on the media (in order to validate its integrity) and takes appropriate action, ranging from a simple block reclamation to the erasure of dangling extent links. This process is merged with the filesystem's normal mount procedure in order to achieve optimal bootstrap timings.

Compression and decompression

For fast and efficient compression/decompression, you can use the `deflate` and `inflator` utilities, which rely on popular deflate/inflate algorithms.

The deflate algorithm combines two algorithms. The first takes care of removing data duplication in files; the second algorithm handles data sequences that appear the most often by giving them shorter symbols. Those two algorithms provide excellent lossless compression of data and executable files. The inflate algorithm simply reverses what the deflate algorithm does.

The `deflate` utility is intended for use with the `filter` attribute for `mkefs`. You can also use it to precompress files intended for a flash filesystem.

The `inflator` resource manager sits in front of the other filesystems that were previously compressed using the `deflate` utility. It can almost double the effective size of the flash memory.

Compressed files can be manipulated with standard utilities such as `cp` or `ftp`—they can display their compressed and uncompressed size with the `ls` utility if used with the proper mountpoint. These features make the management of a compressed flash filesystem seamless to a systems designer.

Flash errors

As flash hardware wears out, its write state-machine may find that it can't write or erase a particular bit cell. When this happens, the error status is propagated to the flash driver so it can take proper action (i.e., mark the bad area and try to write/erase in another place).

This error-handling mechanism is transparent. Note that after several flash errors, all writes and erases that fail will eventually render the flash read-only. Fortunately, this situation shouldn't happen before several years of flash operation. Check your flash specification and analyze your application's data flow to flash in order to calculate its potential longevity or MTBF.

Endian awareness

The FFS3 filesystem is endian-aware, making it portable across different platforms. The optimal approach is to use the `mkefs` utility to select the target's endian-ness.

Utilities

The filesystem supports all the standard POSIX utilities such as `ls`, `mkdir`, `rm`, `ln`, `mv`, and `cp`. There are also some QNX Neutrino utilities for managing the flash:

`flashctl`

Erase, format, and mount flash partitions.

`deflate`

Compress files for flash filesystems.

`mkefs`

Create flash filesystem image files.

System calls

The filesystem supports all the standard POSIX I/O functions such as *open()*, *close()*, *read()*, and *write()*. Special functions such as erasing are supported using the non-POSIX *devctl()* function.

NFS filesystem

The Network File System (NFS) allows a client workstation to perform transparent file access over a network. It allows a client workstation to operate on files that reside on a server across a variety of operating systems. Client file access calls are converted to NFS protocol requests, and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

The Network File System operates in a stateless fashion by using remote procedure calls (RPC) and TCP/IP for its transport. Therefore, to use `fs-nfs2` or `fs-nfs3`, you'll also need to run the TCP/IP client for QNX Neutrino.

Any POSIX limitations in the remote server filesystem will be passed through to the client. For example, the length of filenames may vary across servers from different operating systems. NFS (versions 2 and 3) limits filenames to 255 characters; `mountd` (versions 1 and 3) limits pathnames to 1024 characters.

Although NFS (version 2) is older than POSIX, it was designed to emulate UNIX filesystem semantics and happens to be relatively close to POSIX. If possible, you should use `fs-nfs3` instead of `fs-nfs2`.

CIFS filesystem

Formerly known as SMB, the Common Internet File System (CIFS) allows a client workstation to perform transparent file access over a network to a Windows system, or a UNIX system running an SMB server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

The CIFS protocol makes no attempt to conform to POSIX.

The `fs-cifs` manager uses TCP/IP for its transport. Therefore, to use `fs-cifs`, you'll also need to run the TCP/IP client for the QNX Neutrino RTOS.

Linux Ext2 filesystem

The Ext2 filesystem provides transparent access to Linux disk partitions.

The **fs-ext2.so** implementation supports the standard set of features found in Ext2 versions 0 and 1.

Sparse file support is included in order to be compatible with existing Linux partitions. Other filesystems can only be “stacked” read-only on top of sparse files. There are no such restrictions on normal files.

If an Ext2 filesystem isn't unmounted properly, a filesystem checker is usually responsible for cleaning up the next time the filesystem is mounted. Although the **fs-ext2.so** module is equipped to perform a quick test, it automatically mounts the filesystem as read-only if it detects any significant problems (which should be fixed using a filesystem checker).

Universal Disk Format (UDF) filesystem

The Universal Disk Format (UDF) filesystem provides access to recordable media, such as CD, CD-R, CD-RW, and DVD. It's used for DVD video, but can also be used for backups to CD, and so on. For more information, see *http://osta.org/specs/index.htm*.

The UDF filesystem is supported by the **fs-udf.so** shared object.

In our implementation, UDF filesystems are read-only.

Apple Macintosh HFS and HFS Plus

The Apple Macintosh HFS (Hierarchical File System) and HFS Plus are the filesystems on Apple Macintosh systems.

The **fs-mac.so** shared object provides read-only access to HFS and HFS Plus disks on a QNX Neutrino system. The following variants are recognized: HFS, HFS Plus, HFSJ (HFS Plus with journal), HFS Plus in an HFS wrapper, HFSX (case senstive), and HFS/ISO-9660 hybrid. In QNX Neutrino 7.0.4 or later, you can mount HFSJ filesystems with a hot (or dirty) journal too.

Windows NT filesystem

The NT filesystem is used on Microsoft Windows NT and later.

The **fs-nt.so** shared object provides read-only access to NTFS disks on a QNX Neutrino system.

Inflator pass-through filesystem

QNX Neutrino provides an Inflator *pass-through* filesystem, which is a resource manager that sits in front of other filesystems and inflates files that were previously deflated (using the `deflate` utility).

The `inflator` utility is typically used when the underlying filesystem is a flash filesystem. Using it can almost double the effective size of the flash memory.

If a file is being opened for a read, `inflator` attempts to open the file itself on an underlying filesystem. It reads the first 16 bytes and checks for the signature of a deflated file. If the file was deflated, `inflator` places itself between the application and the underlying filesystem. All reads return the original file data before it was deflated.

From the application's point of view, the file appears to be uncompressed. Random seeks are also supported. If the application does a *stat()* on the file, the size of the inflated file (the original size before it was deflated) is returned.

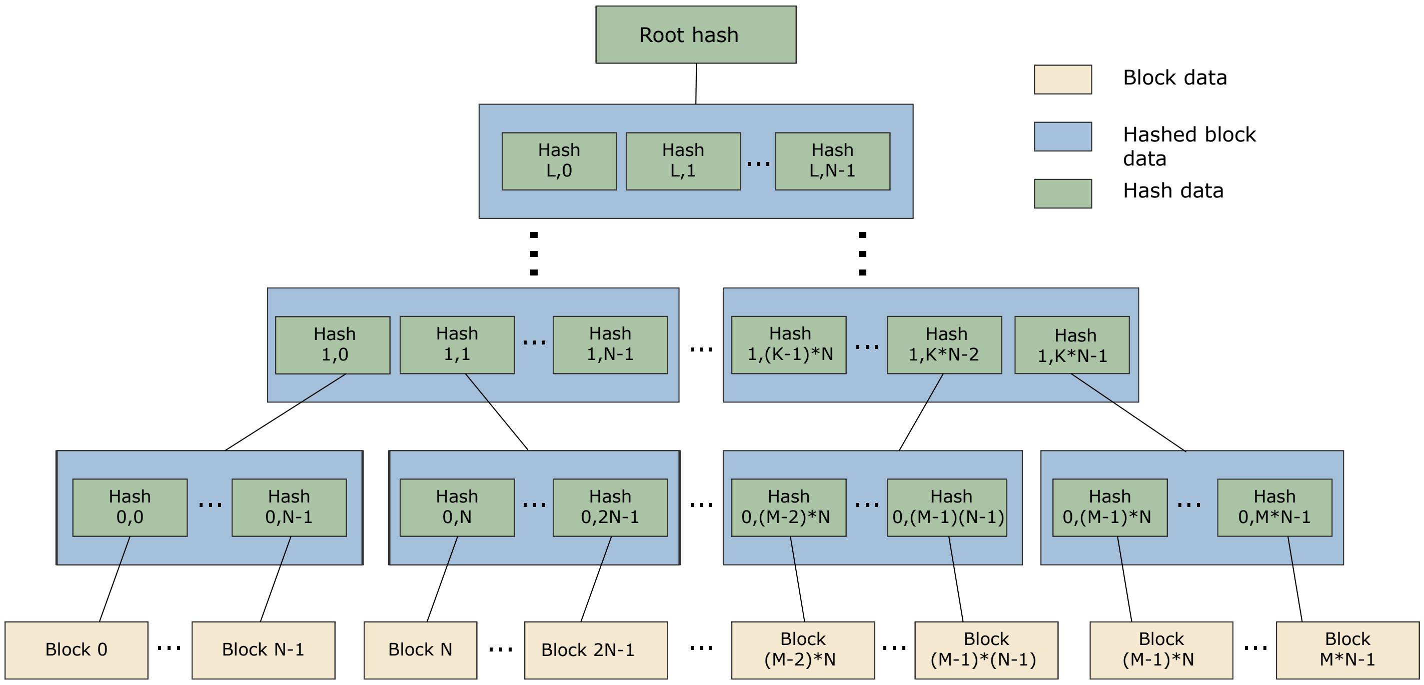
Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

QNX Trusted Disk

QNX Trusted Disk (QTD) devices provide integrity protection of the underlying disk data in secure boot environments. They can extend the secure boot chain up to the core operating system filesystem that stores the critical binaries and configuration files.

The QTD protection mechanism is based on a Merkle tree and is supported by the **fs-qtd.so** shared object. It is the recommended replacement for QNX Neutrino Merkle filesystems.

When building a QTD image, a metadata hash tree is constructed from the blocks of the source filesystem image.



The QTD driver sits between the raw block device and the upper filesystem layer that is supported (for example, a Power-Safe filesystem supported by **fs-qnx6.so**). On read access, it uses the hash tree metadata to verify the integrity of the data before it allows it to be returned to the requester. If the verification fails, an error is returned instead. QTD disk devices are read-only.

The QTD metadata is signed and verified using a key pair. Verifying the signature ensures that the root hash of the tree is valid and hasn't been tampered with. It is the root of the trusted verification mechanism.

The size of the QTD image depends on the chosen block size as well as the chosen digest algorithm. You can use the `mkqfs` utility to generate statistics that describe how much additional space the metadata consumes.

QTD can also be used as a package container solution by mounting files that are themselves QTD images.

Secure hash algorithms

QTD supports the following secure hash algorithms: sha256, sha512.

Signing keys

Refer to the `mkqfs` utility for the supported key types and signing algorithms.

Crypto engines

OpenSSL is the only crypto engine that QTD supports.

Merkle filesystem

Merkle filesystems are a type of block-level validated filesystem, providing integrity protection when they are used in a secure boot environment. QNX recommends that you replace Merkle filesystems with QNX Trusted Disk.

When the Merkle filesystem is built, a metadata hash tree is constructed from the blocks of the source filesystem image.

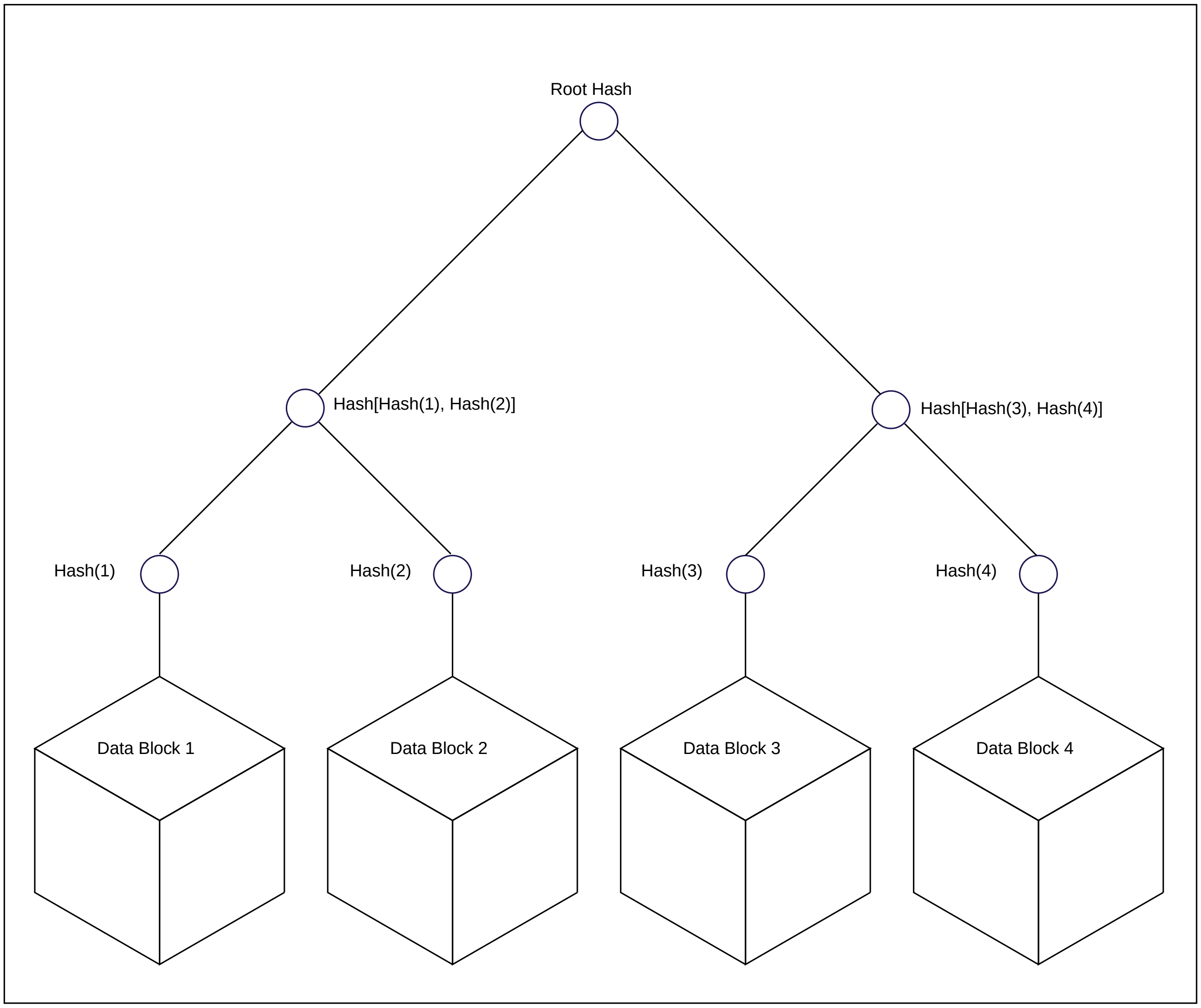


Figure 43: Hash values are created from data blocks

The *Merkle* filesystem filter is supported by the **fsf-merkle.so** shared object.

The Merkle driver sits between the raw block device and the upper filesystem layer that is supported (for example, the Power-Safe filesystem, **fs-qnx6.so**). It verifies the integrity of the data upon read access, using the hash tree metadata, before allowing it to be returned to the requester. In case of a verification failure, an error is returned instead. Merkle filesystems are read-only.

The Merkle hash tree metadata is signed and verified using a key pair. Verifying the signature ensures that the root hash of the tree is valid and hasn't been tampered with. It is the root of the trusted verification mechanism.

The size of the Merkle filesystem image depends on the underlying filesystem block geometry as well as the chosen digest algorithm. The `mkmerklefs` utility can be used to generate statistics of how much extra space is consumed by the metadata.

Secure hash algorithms

Merkle filesystems support the following secure hash algorithms: sha256, sha512.

Signing keys

Signing keys need to be generated as a 2048-bit RSA public/private key pair in PEM format. You can generate a 2048-bit RSA key pair with the following commands:

```
openssl genrsa -out private_key.pem 2048
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Signing can also be handled using a custom utility. For more information, see the options for the `mkmerklefs` utility.

Crypto engines

OpenSSL is the only crypto engine supported by Merkle filesystems on QNX.

Filesystem events

Keeping up with a very "live" filesystem that changes often and quickly is a challenge for the programs that work with it. To help with this challenge, QNX Neutrino includes *filesystem events*.

These events let an application keep track of changes—including the creation and deletion of files and directory, changes of ownership and permissions, mount and unmount operations, and more—in a filesystem that's of interest. The management of filesystem events involves an event manager, an event mechanism, and client event handlers.

The *event manager*, `fsevmgr`, is a process that registers a fixed name in the system namespace (the default is **/dev/fsevents**) and receives the filesystem events, possibly from multiple **io-blk.so** instances, determines which clients need which events, and gives the events to those clients.

The *event mechanism* is part of **io-blk.so**, where any filesystem or parts of the block I/O system can easily use it. This gives the most flexibility since we can coordinate events coming into the filesystem at the system-call level, from within the filesystems, partitions, caching, and even the block device driver if needed.

The mechanism places events into a buffer to eventually be sent to the event manager. This mechanism is limited to an API that reports a locale where the event came from, a unique identifier indicating the event, and properties that further describe the event. The **io-blk.so** library is responsible for packaging each event and sending it to the event manager. The implemention uses a timer to ensure events aren't left sitting in this buffer for an extended period of time.

During startup, **io-blk.so** looks for the event manager. If it isn't loaded, the event mechanism is disabled. Three configuration parameters are provided:

- `fse-device` — the name of the event manager
- `fse-period` — the maximum period of time to delay before sending events to the event manager
- `fse-size` — the size of the buffers holding events

During the loading of the event manager, a DCMD_FSYS_FSEVMGR_CHECK *devctl* command is sent to the various **io-blk.so** mountpoints to notify them that an event manager has been loaded.

Finally a *client event handler* may instantiate a thread that blocks on reads from **/dev/fsevents** or selects the file descriptor for notification once data is available.

Here's an overview of how the client typically interacts with the event manager:

open()

FSE_DEFAULT_MANAGER_NAME (defined in **<sys/fs_events.h>**) describes the default path of the event manager. Keep in mind that it may be overridden by the event manager, and **io-blk.so** may also be changed to look for another event manager.

The client should generally open the event manager in O_RDONLY mode to simply read events. Write mode is necessary only if the client needs to send events. Set the O_NONBLOCK flag if the client is going to poll the event manager.

read()

The client reads events from the Event Manager into an array of bytes. The byte array is filled up to the size required to store whole events. In other words, partial events aren't returned to readers. No assumptions can be made reguarding the alignment of events within the buffers.

FSE_READ_EVENT_S(*pev*, *pbytes*) is provided as a wrapper to *memcpy()* an event from the byte array into an aligned structure. From there, the client can use the other *FSE_** accessor macros to interpret event characteristics.

While the event manager generally has a large queue to hold events, it will inevitably fill and begin to overwrite old events. If any event clients haven't kept up with the addition of new events, those readers will be placed into an overflow state. In other words, if the event manager has begun to overwrite events that a reader hasn't consumed, the next read from that descriptor indicate an error of EOVERFLOW. The subsequent *read()* returns the oldest event.

lseek()

The event manager supports a limited seek functionality. Generally, a client starts reading from the head of the queue and immediately blocks waiting for new events. If a client must evaluate past events, it can use *lseek()*.

A *whence* value of SEEK_SET and an offset of zero position the file descriptor to the oldest event in the queue. Be aware that this condition may cause an EOVERFLOW during a subsequent *read()*.

A *whence* of SEEK_END and an offset of zero position the reader at the end of the queue to read new events.

The client can use event counts as the offset, with either SEEK_SET or SEEK_END, to index some count of events from the head or tail of the queue, respectively.

write()

The client can use *write()* to send events to the event manager. The event manager checks the event data to ensure it's well-formed; on failure, the event manager aborts the writing of all events that were sent in a single write operation.

close()

Once the file descriptor is no longer needed, it may be closed.

The **<sys/fs_events.h>** header file includes everything that's necessary to read and process events from the event manager. For more information, see *FSE_*()* in the *C Library Reference*.

Chapter 10
PPS

The QNX Neutrino Persistent Publish/Subscribe (PPS) service is a small, extensible publish/subscribe service that offers persistence across reboots. It's designed to provide a simple and easy-to-use solution for both publish/subscribe and persistence in embedded systems, answering a need for building loosely connected systems using asynchronous publications and notifications.

With PPS, publishing is asynchronous: the subscriber need not be waiting for the publisher. In fact, the publisher and subscriber rarely know each other; their only connection is an object which has a meaning and purpose for both publisher and subscriber.

The PPS design is in many ways similar to many process control systems where the objects are control values updated by hardware or software. Subscribers can be alarm handling code, displays, and so on. Since there is a single instance of an object, persistence is a natural property that can be applied to it.

Persistence

PPS maintains its objects in memory while it's running.

It will, as required:

- save its objects to persistent storage, either on demand while it's running, or at shutdown
- restore its objects on startup, either immediately, or on first access (deferred loading)

The underlying persistent storage used by PPS relies on a reliable filesystem, such as:

- disk—Power-Safe filesystem
- NAND Flash—ETFS filesystem
- Nor Flash—FFS3 filesystem
- other—a customer-generated filesystem

When PPS starts up, it immediately builds the directory hierarchy from the encoded filenames on the persistent filesystem. It defers loading the objects in the directories until first access to one of the files. This access could be an *open()* call on a PPS object, or a *readdir()* call on the PPS directory.

On shutdown, PPS always saves any modified objects to a persistent filesystem. You can also force PPS to save an object at any time by calling *fsync()* on the object. When PPS saves to a persistent filesystem, it saves all objects to a single directory.

You can set PPS object and attribute qualifiers to have PPS *not* save specific objects or attributes.

PPS objects

PPS uses an object-based system; that is, a system with objects whose properties a publisher can modify. Clients that subscribe to an object receive updates when that object changes—when the publisher has modified it.

PPS objects exist as files with attributes in a special PPS filesystem. By default, PPS objects appear under **/pps**. You can:

- Create directories and populate them with PPS objects by creating files in the directories.
- Use the *open()*, then the *read()* and *write()* functions to query and change PPS objects.
- Use standard utilities as simple debugging tools.

PPS directories can include special objects, such as **.all** and **.notify**, which applications can open to facilitate subscription behavior.

When PPS creates, deletes, or truncates an object (a file or a directory), it places a notification string into the queue of any subscriber or publisher that has open either that object or the **.all** special object for the directory with the modified object. This file can be open in either full or delta mode.

PPS supports pathname open options, and objects and attribute qualifiers. PPS uses pathname open options to apply open options on the file descriptor used to open an object. Object and attribute qualifiers set specific actions to take with an object or attribute; for example, make an object non-persistent, or delete an attribute.

Pathname open options

PPS objects support an extended syntax on the pathnames used to open them. Open options are added as suffixes to the pathname, following a question mark (“?”). That is, the PPS service uses any data that follows a question mark in a pathname to apply open options on the file descriptor used to access the object. Multiple options are separated by question marks.

Object and attribute qualifiers

You can set qualifiers to *read()* and *write()* calls by starting a line containing an object or attribute name with an opening square bracket, followed by a list of single-letter or single-numeral qualifiers and terminated by a closing square bracket.

Publishing

To publish to a PPS object, a publisher simply calls *open()* for the object file with O_WRONLY to publish only, or O_RDWR to publish and subscribe. The publisher can then call *write()* to modify the object's attributes. This operation is non-blocking.

PPS supports multiple publishers that publish to the same PPS object. This capability is required because different publishers may have access to data that applies to different attributes for the same object.

In a multimedia system, for instance, the renderer may be the source of a `time::`*value* attribute, while the HMI may be the source of a `duration::`*value* attribute. A publisher that changes only the `time` attribute will update only that attribute when it writes to the object. It will leave the other attributes unchanged.

Subscribing

PPS clients can subscribe to multiple objects, and PPS objects can have multiple subscribers. When a publisher changes an object, all clients subscribed to that object are informed of the change.

To subscribe to an object, a client simply calls *open()* for the object with O_RDONLY to subscribe only, or O_RDWR to publish and subscribe. The subscriber can then query the object with a *read()* call.

A subscriber can open an object in full mode, in delta mode, or in full and delta modes at the same time. The figure below illustrates the different information sent to subscribers who open a PPS object in full mode and in delta mode.

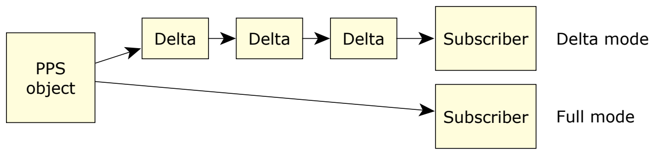


Figure 44: PPS full and delta subscription modes.

In all cases, PPS maintains persistent objects with states—there's always an object. The mode used to open an object *doesn't* change the object; it determines only the subscriber's view of the object.

Full mode

In full mode (the default), the subscriber always receives a single, consistent version of the entire object as it exists at the moment when it is requested.

If a publisher changes an object several times before a subscriber asks for it, the subscriber receives the state of the object at the time of asking *only*. If the object changes again, the subscriber is notified again of the change. Thus, in full mode, the subscriber may miss multiple changes to an object—changes to the object that occur before the subscriber asks for it.

Delta mode

In delta mode, a subscriber receives only the changes (but all the changes) to an object's attributes. On the first read, since a subscriber knows nothing about the state of an object, PPS assumes everything has changed. Therefore, a subscriber's first read in delta mode returns all attributes for an object, while subsequent reads return only the changes since that subscriber's previous read. Thus, in delta mode, the subscriber always receives all changes to an object.

PPS uses directories as a natural grouping mechanism to simplify and make more efficient the task of subscribing to multiple objects. Subscribers can open multiple objects, either by calling *open()* then *select()* on the objects, or, more easily, by opening the special **.all** object which merges all objects in its directory.

PPS provides a mechanism to associate a set of file descriptors with a notification group. This mechanism allows you to read only the PPS special notification object to receive notification of changes to all objects associated with a notification group.

Chapter 11 Character I/O

A key requirement of any realtime operating system is high-performance character I/O.

Character devices can be described as devices to which I/O consists of a sequence of bytes transferred serially, as opposed to block-oriented devices (e.g., disk drives).

As in the POSIX and UNIX tradition, these character devices are located in the OS pathname space under the **/dev** directory. For example, a serial port to which a modem or terminal could be connected might appear in the system as:

/dev/ser1

Typical character devices found on PC hardware include:

- serial ports
- parallel ports
- text-mode consoles
- pseudo terminals (ptys)

Programs access character devices using the standard *open()*, *close()*, *read()*, and *write()* API functions. Additional functions are available for manipulating other aspects of the character device, such as baud rate, parity, flow control, etc.

Since it's common to run multiple character devices, they have been designed as a family of drivers and a library called `io-char` to maximize code reuse.

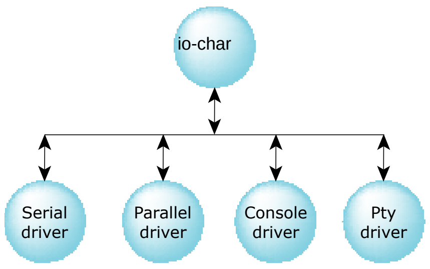


Figure 45: The `io-char` module is implemented as a library.

As shown in this diagram, `io-char` is implemented as a library. The `io-char` module contains all the code to support POSIX semantics on the device. It also contains a significant amount of code to implement character I/O features beyond POSIX but desirable in a realtime system. Since this code is in the common library, all drivers inherit these capabilities.

The driver is the executing process that calls into the library. In operation, the driver starts first and invokes `io-char`. The drivers themselves are just like any other QNX Neutrino process and can run at different priorities according to the nature of the hardware being controlled and the client's requesting service.

Once a single character device is running, the memory cost of adding additional devices is minimal, since only the code to implement the new driver structure would be new.

Driver/io-char communication

The io-char library manages the flow of data between an application and the device driver. Data flows between io-char and the driver through a set of memory queues associated with each character device.

Three queues are used for each device. Each queue is implemented using a first-in, first-out (FIFO) mechanism.

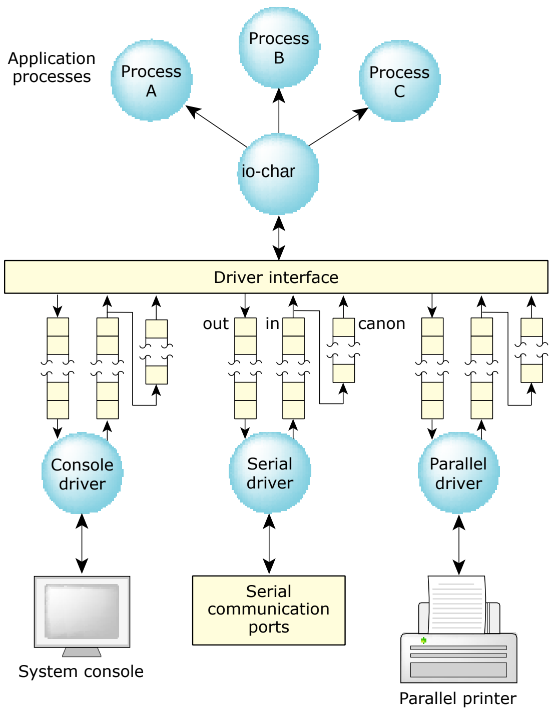


Figure 46: Device I/O in the QNX Neutrino RTOS.

Received data is placed into the *raw* input queue by the driver and is consumed by io-char only when application processes request data. (For details on raw versus edited or canonical input, see the section "*Input modes*" later in this chapter.)

Interrupt handlers within drivers typically call a trusted library routine within io-char to add data to this queue—this ensures a consistent input discipline and minimizes the responsibility of the driver (and effort required to create new drivers).

The io-char module places output data into the output queue to be consumed by the driver as characters are physically transmitted to the device. The module calls a trusted routine within the driver each time new data is added so it can "kick" the driver into operation (in the event that it was idle). Since output queues are used, io-char implements *write-behind* for all character devices. Only when the output buffers are full will io-char cause a process to block while writing.

The canonical queue is managed entirely by io-char and is used while processing input data in *edited* mode. The size of this queue determines the maximum edited input line that can be processed for a particular device.

The sizes of these queues are configurable using command-line options. Default values are usually more than adequate to handle most hardware configurations, but you can “tune” these to reduce overall system memory requirements, to accommodate unusual hardware situations, or to handle unique protocol requirements.

Device drivers simply add received data to the raw input queue or consume and transmit data from the output queue. The `io-char` module decides when (and if) output transmission is to be suspended, how (and if) received data is echoed, etc.

Device control

Low-level device control is implemented using the *devctl()* call.

The POSIX terminal control functions are layered on top of *devctl()* as follows:

tcgetattr()

Get terminal attributes.

tcsetattr()

Set terminal attributes.

tcgetpgrp()

Get ID of process group leader for a terminal.

tcsetpgrp()

Set ID of process group leader for a terminal.

tcsendbreak()

Send a break condition.

tcflow()

Suspend or restart data transmission/reception.

QNX Neutrino extensions

The QNX Neutrino extensions to the terminal control API are as follows:

tcdropline()

Initiate a disconnect. For a serial device, this will pulse the DTR line.

tcinject()

Inject characters into the canonical buffer.

The `io-char` module acts directly on a common set of *devctl()* commands supported by most drivers. Applications send device-specific *devctl()* commands through `io-char` to the drivers.

Input modes

Each device can be in a *raw* or *edited* input mode.

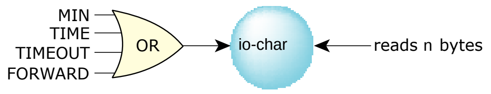
Raw input mode

In raw mode, `io-char` performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data.

Fullscreen programs and serial communications programs are examples of applications that use a character device in raw mode.

In raw mode, each character is received into the raw input buffer by the interrupt handler. When an application requests data from the device, it can specify under what conditions an input request is to be satisfied. Until the conditions are satisfied, the interrupt handler won't signal the driver to run, and the driver won't return any data to the application. The normal case of a simple read by an application would block until at least one character was available.

The following diagram shows the full set of available conditions:



MIN	Respond when at least this number of characters arrives.
TIME	Respond if a pause in the character stream occurs.
TIMEOUT	Respond if an overall amount of time passes.
FORWARD	Respond if a framing character arrives.

Figure 47: Conditions for satisfying an input request.

In the case where multiple conditions are specified, the read will be satisfied when any one of them is satisfied.

MIN

The qualifier *MIN* is useful when an application has knowledge of the number of characters it expects to receive.

Any protocol that knows the character count for a frame of data can use *MIN* to wait for the entire frame to arrive. This significantly reduces IPC and process scheduling. *MIN* is often used in conjunction with *TIME* or *TIMEOUT*. *MIN* is part of the POSIX standard.

TIME

The qualifier *TIME* is useful when an application is receiving streaming data and wishes to be notified when the data stops or pauses. The pause time is specified in 1/10ths of a second. *TIME* is part of the POSIX standard.

TIMEOUT

The qualifier *TIMEOUT* is useful when an application has knowledge of how long it should wait for data before timing out. The timeout is specified in 1/10ths of a second.

Any protocol that knows the character count for a frame of data it expects to receive can use *TIMEOUT*. This in combination with the baud rate allows a reasonable guess to be made when data should be available. It acts as a deadman timer to detect dropped characters. It can also be used in interactive programs with user input to time out a read if no response is available within a given time.

TIMEOUT is a QNX Neutrino extension and is not part of the POSIX standard.

FORWARD

The qualifier *FORWARD* is useful when a protocol is delimited by a special framing character. For example, the PPP protocol used for TCP/IP over a serial link starts and ends its packets with a framing character. When used in conjunction with *TIMEOUT*, the *FORWARD* character can greatly improve the efficiency of a protocol implementation. The protocol process will receive complete frames, rather than character by character. In the case of a dropped framing character, *TIMEOUT* or *TIME* can be used to quickly recover.

This greatly minimizes the amount of IPC work for the OS and results in a much lower processor utilization for a given TCP/IP data rate. It's interesting to note that PPP doesn't contain a character count for its frames. Without the data-forwarding character, an implementation might be forced to read the data one character at a time.

FORWARD is a QNX Neutrino extension and is not part of the POSIX standard.

The ability to “push” the processing for application notification into the service-providing components of the OS reduces the frequency with which user-level processing must occur. This minimizes the IPC work to be done in the system and frees CPU cycles for application processing. In addition, if the application implementing the protocol is executing on a different network node than the communications port, the number of network transactions is also minimized.

For intelligent, multiport serial cards, the data-forwarding character recognition can also be implemented within the intelligent serial card itself, thereby significantly reducing the number of times the card must interrupt the host processor for interrupt servicing.

Edited input mode

In edited mode, `io-char` performs line-editing operations on each received character. Only when a line is “completely entered”—typically when a carriage return (CR) is received—will the line of data be made available to application processes. This mode of operation is often referred to as *canonical* or sometimes “cooked” mode.

Most nonfullscreen applications run in edited mode, because this allows the application to deal with the data a line at a time, rather than have to examine each character received, scanning for an end-of-line character.

In edited mode, each character is received into the raw input buffer by the interrupt handler. Unlike raw mode where the driver is scheduled to run only when some input conditions are met, the interrupt handler will schedule the driver on every received character.

There are two reasons for this. First, edited input mode is rarely used for high-performance communication protocols. Second, the work of editing is significant and not suitable for an interrupt handler.

When the driver runs, code in `io-char` will examine the character and apply it to the canonical buffer in which it's building a line. When a line is complete and an application requests input, the line will be transferred from the canonical buffer to the application—the transfer is direct from the canonical buffer to the application buffer without any intervening copies.

The editing code correctly handles multiple pending input lines in the canonical buffer and allows partial lines to be read. This can happen, for example, if an application asked only for 1 character when a 10-character line was available. In this case, the next read will continue where the last one left off.

The `io-char` module provides a rich set of editing capabilities, including full support for moving over the line with cursor keys and for changing, inserting, or deleting characters. Here are some of the more common capabilities:

LEFT

Move the cursor one character to the left.

RIGHT

Move the cursor one character to the right.

HOME

Move the cursor to the beginning of the line.

END

Move the cursor to the end of the line.

ERASE

Erase the character to the left of the cursor.

DEL

Erase the character at the current cursor position.

KILL

Erase the entire input line.

UP

Erase the current line and recall a previous line.

DOWN

Erase the current line and recall the next line.

INS

Toggle between insert mode and typeover mode (every new line starts in insert mode).

Line-editing characters vary from terminal to terminal. The console always starts out with a full set of editing keys defined.

If a terminal is connected via a serial channel, you need to define the editing characters that apply to that particular terminal. To do this, you can use the `stty` utility. For example, if you have an ANSI terminal connected to a serial port (called **/dev/ser1**), you would use the following command to extract the appropriate editing keys from the **terminfo** database and apply them to **/dev/ser1**:

```
stty term=ansi </dev/ser1
```

Device subsystem performance

The flow of events within the device subsystem is engineered to minimize overhead and maximize throughput when a device is in *raw* mode. To accomplish this, the following rules are used:

- Interrupt handlers place received data directly into a memory queue. Only when a read operation is pending, *and* that read operation can be satisfied, will the interrupt handler schedule the driver to run. In all other cases, the interrupt simply returns. Moreover, if `io-char` is already running, no scheduling takes place, since the availability of data will be noticed without further notification.
- When a read operation is satisfied, the driver replies to the application process *directly* from the raw input buffer into the application's receive buffer. The net result is that the data is copied only once.

These rules—coupled with the extremely small interrupt and scheduling latencies inherent within the OS—result in a very lean input model that provides POSIX conformance together with extensions suitable to the realtime requirements of protocol implementations.

Console devices

System consoles (with VGA-compatible graphics chips in *text* mode) are managed by the `devc-con` or `devc-con-hid` driver. The video display card/screen and the system keyboard are collectively referred to as the *physical console*.

The `devc-con` or `devc-con-hid` driver permits multiple sessions to be run concurrently on a physical console by means of *virtual consoles*. The `devc-con` console driver process typically manages more than one set of I/O queues to `io-char`, which are made available to user processes as a set of *character devices* with names like **/dev/con1**, **/dev/con2**, etc. From the application's point of view, there “really are” multiple consoles available to be used.

Of course, there's only one *physical console* (screen and keyboard), so only *one* of these virtual consoles is actually displayed at any one time. The keyboard is “attached” to whichever virtual console is currently visible.

Terminal emulation

The console drivers emulate an ANSI terminal.

Serial devices

Serial communication channels are managed by the `devc-ser*` family of driver processes. These drivers can manage more than one physical channel and provide character devices with names such as **/dev/ser1**, **/dev/ser2**, etc.

When `devc-ser*` is started, command-line arguments can specify which—and how many—serial ports are installed. On a PC-compatible system, this will typically be the two standard serial ports often referred to as **com1** and **com2**. The `devc-ser*` driver directly supports most nonintelligent multiport serial cards.

QNX Neutrino includes various serial drivers (e.g., `devc-ser8250`). For details, see the `devc-ser*` entries in the *Utilities Reference*.

The `devc-ser*` drivers support hardware flow control (except under edited mode) provided that the hardware supports it. Loss of carrier on a modem can be programmed to deliver a SIGHUP signal to an application process (as defined by POSIX).

Pseudo terminal devices (ptys)

Pseudo terminals are managed by the `devc-pty` driver.

Command-line arguments to `devc-pty` specify the number of pseudo terminals to create.

A pseudo terminal (pty) is a *pair* of character devices: a master device and a slave device. The slave device provides an interface identical to that of a tty device as defined by POSIX. However, while other tty devices represent hardware devices, the slave device instead has another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input; anything written on the slave device is presented as input to the master device. As a result, pseudo-ttys can be used to connect processes that would otherwise expect to be communicating with a character device.

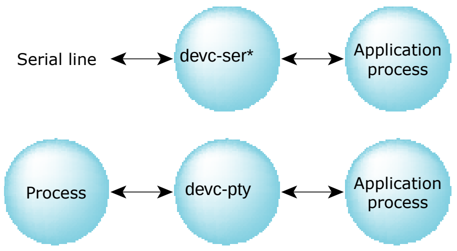


Figure 48: Pseudo-ttys.

Ptys are routinely used to create pseudo-terminal interfaces for programs such as `telnet`, which uses TCP/IP to provide a terminal session to a remote system.

Chapter 12
Networking Architecture

As with other service-providing processes in the QNX Neutrino RTOS, the networking services execute outside the kernel. Developers are presented with a single unified interface, regardless of the configuration and number of networks involved.

This architecture allows:

- network drivers to be started and stopped *dynamically*
- Qnet and other protocols to run together in any combination

Our native network subsystem consists of the network manager executable (`io-pkt-v4-hc` or `io-pkt-v6-hc`), plus one or more shared library modules. These modules can include protocols (e.g., **lsm-qnet.so**) and drivers (e.g., `devnp-speedo.so`).

Network manager (`io-pkt*`)

The `io-pkt*` component is the active executable within the network subsystem. Acting as a kind of packet redirector/multiplexer, `io-pkt*` is responsible for loading protocol and driver modules based on the configuration given to it on its command line (or via the `mount` command after it's started).

Employing a zero-copy architecture, the `io-pkt*` executable efficiently loads multiple networking protocols or drivers (e.g., **lsm-qnet.so**) *on the fly*—these modules are shared objects that install into `io-pkt*`.

The `io-pkt` stack is very similar in architecture to other component subsystems inside the operating system. At the bottom layer are drivers that provide the mechanism for passing data to and receiving data from the hardware. The drivers hook into a multi-threaded layer-2 component (that also provides fast forwarding and bridging capability) that ties them together and provides a unified interface for directing packets into the protocol-processing components of the stack. This includes, for example, handling individual IP and upper-layer protocols such as TCP and UDP.

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

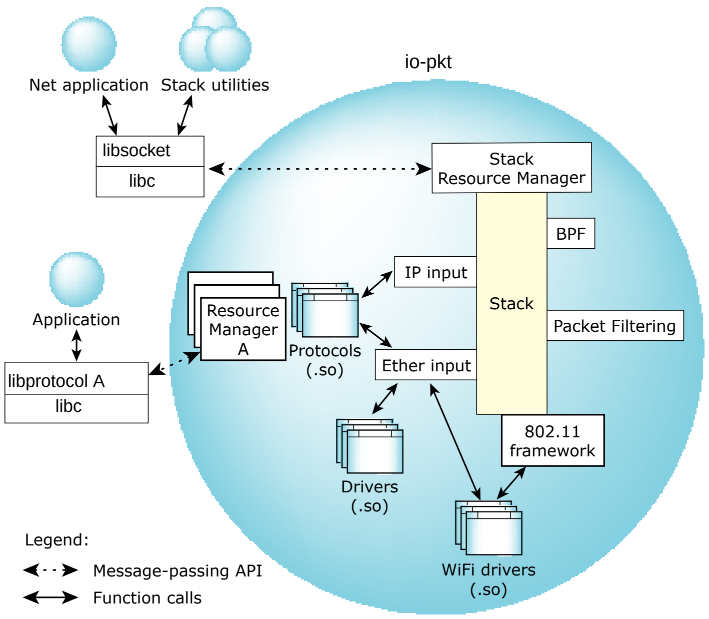


Figure 49: A detailed view of the `io-pkt` architecture.

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

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

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

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

Berkeley Packet Filter (BPF) interface

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

Packet Filter (PF) interface

A read/write/modify/block interface that gives complete control over which packets are received by or transmitted from the upper layers.

For more information, see the Packet Filtering and Firewalling chapter of the QNX Neutrino Core Networking *User's Guide*.

Threading model

The default mode of operation is for `io-pkt` to create one thread per CPU.

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

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

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

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

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

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

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

Protocol module

The networking protocol module is responsible for implementing the details of a particular protocol (e.g., Qnet).

Each protocol component is packaged as a shared object (e.g., **lsm-qnet.so**). One or more protocol components may run concurrently.

For example, the following line from a buildfile shows `io-pkt-v4-hc` loading the Qnet protocol via its `-p` *protocol* command-line option:

```
io-pkt-v4-hc -d abc100 -pqnet
```

The `io-pkt*` managers include the TCP/IP stack.

Qnet is the QNX Neutrino native networking protocol. Its main purpose is to extend the OS's powerful message-passing IPC *transparently* over a network of microkernels.

Qnet also provides Quality of Service policies to help ensure reliable network transactions.

For more information on the Qnet and TCP/IP protocols, see the following chapters in this book:

- *Native Networking (Qnet)*
- *TCP/IP Networking*

Driver module

The network driver module is responsible for managing the details of a particular network adaptor (e.g., an NE-2000 compatible Ethernet controller). Each driver is packaged as a shared object and installs into the `io-pkt*` component.

Once `io-pkt*` is running, you can dynamically load drivers at the command line using the `mount` command.

For example, the following commands start `io-pkt-v6-hc` and then `mount` the driver for the fictitious ABC100 chip set adapter:

```
io-pkt-v6-hc
mount -T io-pkt devnp-abc100.so
```

All network device drivers are shared objects whose names are of the form **devnp-*driver*.so**.

Once the shared object is loaded, `io-pkt*` initializes it. The driver and `io-pkt*` are then effectively bound together—the driver calls into `io-pkt*` (for example when packets arrive from the interface), and `io-pkt*` calls into the driver (for example when packets need to be sent from an application to the interface).

To unload a driver, use the `ifconfig destroy` command. For example:

```
ifconfig abc0 destroy
```

For more information on network device drivers, see their individual utility pages (`devnp-*`) in the *Utilities Reference*.

Chapter 13
Native Networking (Qnet)

In the Interprocess Communication (IPC) chapter earlier in this manual, we described message passing in the context of a single node. But the true power of the QNX Neutrino RTOS lies in its ability to take the message-passing paradigm and extend it *transparently* over a network of microkernels. This chapter describes QNX Neutrino native networking (via the Qnet protocol).

You can have at most one instance of Qnet running on a node, even if you're running more than one instance of `io-pkt`.

QNX Neutrino distributed

At the heart of QNX Neutrino native networking is the Qnet protocol, which is deployed as a network of tightly coupled trusted machines.

Qnet lets these machines share their resources efficiently with little overhead. Using Qnet, you can use the standard OS utilities (`cp`, `mv`, and so on) to manipulate files anywhere on the Qnet network as if they were on your machine. In addition, the Qnet protocol doesn't do any authentication of remote requests; files are protected by the normal permissions that apply to users and groups. Besides files, you can also access and start/stop *processes*, including managers, that reside on any machine on the Qnet network.

The distributed processing power of Qnet lets you do the following tasks efficiently:

- Access your remote filesystem.
- Scale your application with unprecedented ease.
- Write applications using a collection of cooperating processes that communicate transparently with each other using QNX Neutrino message-passing.
- Extend your application easily beyond a single processor or SMP machine to several single-processor machines and distribute your processes among those CPUs.
- Divide your large application into several processes, where each process can perform different functions. These processes coordinate their work using message passing.
- Take advantage of Qnet's inherent remote procedure call functionality.

Moreover, since Qnet extends QNX Neutrino message passing over the network, other forms of IPC (e.g., signals, message queues, named semaphores) also work over the network.

To understand how network-wide IPC works, consider two processes that wish to communicate with each other: a client process and a server process (in this case, the serial port manager process). In the single-node case, the client simply calls *open()*, *read()*, *write()*, etc. As we'll see shortly, a high-level POSIX call such as *open()* actually entails message-passing kernel calls “underneath” (*ConnectAttach()*, *MsgSend()*, etc.). But the client doesn't need to concern itself with those functions; it simply calls *open()*.

```
fd = open("/dev/ser1", O_RDWR,...); /* Open a serial device */
```

Now consider the case of a simple network with two machines—one contains the client process, the other contains the server process.

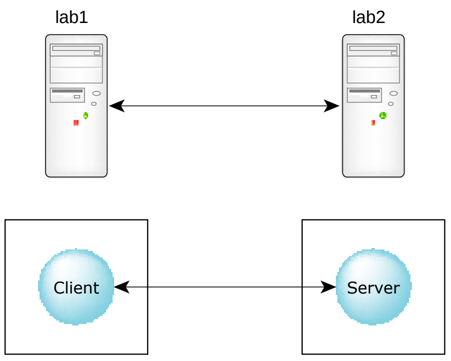


Figure 50: A simple network where the client and server reside on separate machines.

The code required for client-server communication is *identical* to the code in the single-node case, but with one important exception: *the pathname*. The pathname will contain a prefix that specifies the node that the service (**/dev/ser1**) resides on. As we'll see later, this prefix will be translated into a node descriptor for the lower-level *ConnectAttach()* kernel call that will take place. Each node in the network is assigned a node descriptor, which serves as the only visible means to determine whether the OS is running as a network or standalone.

For more information on node descriptors, see the Transparent Distributed Processing with Qnet chapter of the QNX Neutrino *Programmer's Guide*.

Name resolution and lookup

When you run Qnet, the pathname space of all the nodes in your Qnet network is added to yours. Recall that a pathname is a symbolic name that tells a program where to find a file within the directory hierarchy based at root (**/**).

The pathname space of remote nodes will appear under the prefix **/net** (the directory created by the Qnet protocol manager, **lsm-qnet.so**, by default).

For example, remote **node1** would appear as:

```
/net/node1/dev/socket
/net/node1/dev/ser1
/net/node1/home
/net/node1/bin
...
```

So with Qnet running, you can now open pathnames (files or managers) on other remote Qnet nodes, just as you open files locally on your own node. This means you can access regular files or manager processes on other Qnet nodes as if they were executing on your local node.

Recall our *open()* example above. If you wanted to open a serial device on **node1** instead of on your local machine, you simply specify the path:

```
fd = open("/net/node1/dev/ser1", O_RDWR, ...); /* Open a serial device on node1 */
```

For client-server communications, how does the client know what node descriptor to use for the server?

The client uses the filesystem's *pathname space* to "look up" the server's address. In the single-machine case, the result of that lookup will be a node descriptor, a process ID, and a channel ID. In the networked case, the results are the same—the only difference will be the *value* of the node descriptor.

If node descriptor is:	Then the server is:
0 (or ND_LOCAL_NODE)	Local (i.e., "this node")
Nonzero	Remote

File descriptor (connection ID)

The practical result in both the local and networked case is that when the client connects to the server, the client gets a file descriptor (or connection ID in the case of kernel calls such as *ConnectAttach()*). This file descriptor is then used for all subsequent message-passing operations. Note that from the client's perspective, the file descriptor is identical for both the local and networked case.

Behind a simple *open()*

Let's return to our *open()* example. Suppose a client on one node (**lab1**) wishes to use the serial port (**/dev/ser1**) on another node (**lab2**). The client will effectively perform an *open()* on the pathname **/net/lab2/dev/ser1**.

The following diagram shows the steps involved when the client *open()*'s **/net/lab2/dev/ser1**:

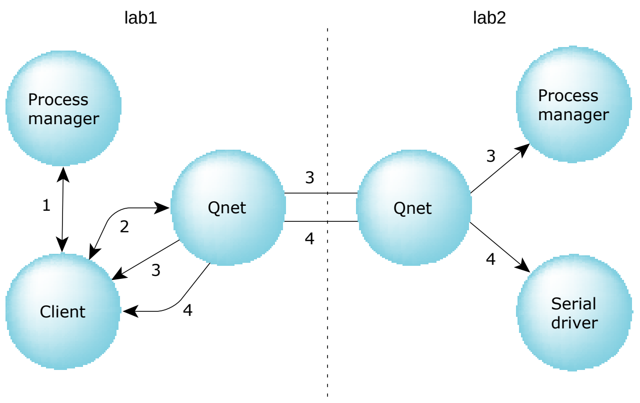


Figure 51: A client-server message pass across the network.

Here are the interactions:

1. A message is sent from the client to its local process manager, effectively asking who should be contacted to resolve the pathname **/net/lab2/dev/ser1**.

   Since the native network manager (**lsm-qnet.so**) has taken over the entire **/net** namespace, the process manager returns a redirect message, saying that the client should contact the local network manager for more information.

2. The client then sends a message to the local network manager, again asking who should be contacted to resolve the pathname.

   The local network manager then replies with another redirect message, giving the node descriptor, process ID, and channel ID of the process manager on node **lab2**—effectively deferring the resolution of the request to node **lab2**.

3. The client then creates a connection to the process manager on node **lab2**, once again asking who should be contacted to resolve the pathname.

   The process manager on node **lab2** returns another redirect, this time with the node descriptor, channel ID, and process ID of the serial driver on its own node.

4. The client creates a connection to the serial driver on node **lab2**, and finally gets a connection ID that it can then use for subsequent message-passing operations.

   After this point, from the client's perspective, message passing to the connection ID is identical to the local case. Note that all further message operations are now direct between the client and server.

The key thing to keep in mind here is that the client isn't aware of the operations taking place; these are all handled by the POSIX *open()* call. As far as the client is concerned, it performs an *open()* and gets back a file descriptor (or an error indication).

In each subsequent name-resolution step, the request from the client is stripped of already-resolved name components; this occurs automagically within the resource manager framework. This means that in step 2 above, the relevant part of the request is **lab2/dev/ser1** from the perspective of the local network manager. In step 3, the relevant part of the request has been stripped to just **dev/ser1**, because that's all that **lab2**'s process manager needs to know. Finally, in step 4, the relevant part of the request is simply **ser1**, because that's all the serial driver needs to know.

Global Name Service (GNS)

In the examples shown so far, remote services or files are located on *known* nodes or at known pathnames. For example, the serial port on **lab1** is found at **/net/lab1/dev/ser1**.

GNS allows you to locate services via an arbitrary name wherever the service is located, whether on the local system or on a remote node. For example, if you wanted to locate a modem on the network, you could simply look for the name “modem.” This would cause the GNS server to locate the “modem” service, instead of using a static path such as **/net/lab1/dev/ser1**. The GNS server can be deployed such that it services all or a portion of your Qnet nodes. And you can have redundant GNS servers.

Network naming

As mentioned earlier, the pathname prefix **/net** is the most common name that **lsm-qnet.so** uses.

In resolving names in a network-wide pathname space, the following terms come into play:

node name

A character string that identifies the node you're talking to. Note that a node name *can't* contain slashes or dots. In the example above, we used **lab2** as one of our node names. The default is fetched via *confstr()* with the _CS_HOSTNAME parameter.

node domain

A character string that's “tacked” onto the node name by **lsm-qnet.so**. Together the node name and node domain *must* form a string that's unique for all nodes that are talking to each other. The default is fetched via *confstr()* with the _CS_DOMAIN parameter.

***fully qualified node name* (FQNN)**

The string formed by tacking the node name and node domain together. For example, if the node name is **lab2** and the node domain name is **qnx.com**, the resulting FQNN would be **lab2.qnx.com**.

network directory

A directory in the pathname space implemented by **lsm-qnet.so**. Each network directory (there can be more than one on a node) has an associated node domain. The default is **/net**, as used in the examples in this chapter.

name resolution

The process by which **lsm-qnet.so** converts an FQNN to a list of destination addresses that the transport layer knows how to get to.

name resolver

A piece of code that implements one method of converting an FQNN to a list of destination addresses. Each network directory has a list of name resolvers that are applied in turn to attempt to resolve the FQNN. The default is `en_ionet` (see the next section).

Quality of Service **(QoS)**

A definition of connectivity between two nodes. The default QoS is `loadbalance` (see the section on *QoS* later in this chapter.)

Resolvers

The following *resolver*s are built into the network manager:

- `en_ionet`—broadcast requests for name resolution on the LAN (similar to the TCP/IP ARP protocol). This is the default.
- `dns`—take the node name, add a dot (`.`) followed by the node domain, and send the result to the TCP/IP *gethostbyname()* function.
- `file`—search for accessible nodes, including the relevant network address, in a static file.

Redundant Qnet: Quality of Service (QoS) and multiple paths

Quality of Service (QoS) is an issue that often arises in high-availability networks as well as realtime control systems.

In the Qnet context, QoS really boils down to *transmission media selection*—in a system with two or more network interfaces, Qnet will choose which one to use according to the policy you specify.

If you have only a single network interface, the QoS policies don't apply at all.

QoS policies

Qnet supports transmission over *multiple networks* and provides several policies for specifying how Qnet should select a network interface for transmission.

These Quality of Service policies include:

`loadbalance` (the default)

Qnet is free to use all available network links, and will share transmission equally among them.

`preferred`

Qnet uses one specified link, ignoring all other networks (unless the preferred one fails).

`exclusive`

Qnet uses one—and only one—link, ignoring all others, even if the exclusive link fails.

To fully benefit from Qnet's QoS, you need to have physically separate networks. For example, consider a network with two nodes and a hub, where each node has two connections to the hub:

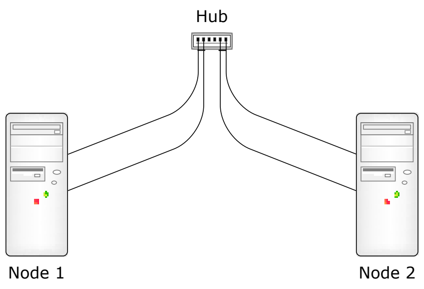


Figure 52: Qnet and a single network.

If the link that's currently in use fails, Qnet detects the failure, but doesn't switch to the other link because both links go to the same hub. It's up to the application to recover from the error; when the application reestablishes the connection, Qnet switches to the working link.

Now, consider the same network, but with two hubs:

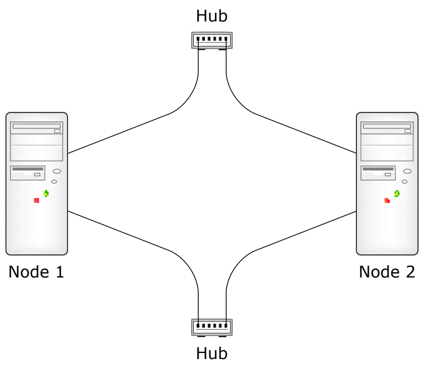


Figure 53: Qnet and physically separate networks.

If the networks are physically separate and a link fails, Qnet automatically switches to another link, depending on the QoS that you chose. The application isn't aware that the first link failed.

You can use the `tx_retries` option to **lsm-qnet.so** to limit the number of times that Qnet retries a transmission, and hence control how long Qnet waits before deciding that a link has failed. Note that if the number of retries is too low, Qnet won't tolerate any lost packets and may prematurely decide that a link is down.

Let's look in more detail at the QoS policies:

`loadbalance`

Qnet decides which links to use for sending packets, depending on current load and link speeds as determined by `io-pkt*`. A packet is queued on the link that can deliver the packet the soonest to the remote end. This effectively provides greater bandwidth between nodes when the links are up (the bandwidth is the sum of the bandwidths of all available links), and allows a graceful degradation of service when links fail.

If a link does fail, Qnet will switch to the next available link. This switch takes a few seconds *the first time*, because the network driver on the bad link will have timed out, retried, and finally died. But once Qnet “knows” that a link is down, it will *not* send user data over that link.

While load-balancing among the live links, Qnet will send periodic maintenance packets on the failed link in order to detect recovery. When the link recovers, Qnet places it back into the pool of available links.

`preferred`

With this policy, you specify a preferred link to use for transmissions. Qnet will use only that one link until it fails. If your preferred link fails, Qnet will then turn to the other available links and resume transmission, using the `loadbalance` policy.

Once your preferred link is available again, Qnet will again use only that link, ignoring all others (unless the preferred link fails).

`exclusive`

You use this policy when you want to lock transmissions to only one link. Regardless of how many other links are available, Qnet will latch onto the one interface you specify. And if that exclusive link fails, Qnet will *NOT* use any other link.

Why would you want to use the `exclusive` policy? Suppose you have two networks, one much faster than the other, and you have an application that moves large amounts of data. You might want to restrict transmissions to only the fast network in order to avoid swamping the slow network under failure conditions.

The `loadbalance` QoS policy is the default.

Specifying QoS policies

You specify the QoS policy as part of the pathname. For example, to access **/net/lab2/dev/ser1** with a QoS of `exclusive`, you could use the following pathname:

```
/net/lab2~exclusive:en0/dev/ser1
```

The QoS parameter always begins with a tilde (~) character. Here we're telling Qnet to lock onto the `en0` interface exclusively, *even if it fails*.

Symbolic links

You can set up symbolic links to the various "QoS-qualified" pathnames:

```
ln -sP /net/lab2~preferred:en1 /remote/sql_server
```

This assigns an "abstracted" name of **/remote/sql_server** to the node **lab2** with a preferred QoS (i.e., over the `en1` link).

You can't create symbolic links inside **/net** because Qnet takes over that namespace.

Abstracting the pathnames by one level of indirection gives you multiple servers available in a network, all providing the same service. When one server fails, the abstract pathname can be "remapped" to point to the pathname of a different server. For example, if **lab2** failed, then a monitoring program could detect this and effectively issue:

```
rm /remote/sql_server
ln -sP /net/lab1 /remote/sql_server
```

This would remove **lab2** and reassign the service to **lab1**. The real advantage here is that applications can be coded based on the abstract "service name" rather than be bound to a specific node name.

Examples

Let's look at a few examples of how you'd use the network manager.

The QNX Neutrino native network manager **lsm-qnet.so** is actually a shared object that installs into the executable `io-pkt*`.

Local networks

If you're using the QNX Neutrino RTOS on a small LAN, you can use just the default `en_ionet` resolver. When a node name that's currently unknown is being resolved, the resolver will broadcast the name request over the LAN, and the node that has the name will respond with an identification message. Once the name's been resolved, it's cached for future reference.

Since `en_ionet` is the default resolver when you start **lsm-qnet.so**, you can simply issue commands like:

```
ls /net/lab2/
```

If you have a machine called “**lab2**” on your LAN, you'll see the contents of its root directory.

Remote networks

Qnet uses DNS (Domain Name System) when resolving remote names. To use **lsm-qnet.so** with DNS, you specify this resolver on `mount`'s command line:

For security reasons, you should have a firewall set up on your network before connecting to the Internet. For more information, see **pf-faq** at *ftp://ftp3.usa.openbsd.org/pub/OpenBSD/doc/* in the OpenBSD documentation.

```
mount -Tio-pkt -o"mount=:,resolve=dns,mount=.com:.net:.edu" /lib/dll/lsm-qnet.so
```

In this example, Qnet will use *both* its native `en_ionet` resolver (indicated by the first `mount=` command) and DNS for resolving remote names.

Note that we've specified several types of domain names (`mount=.com:.net:.edu`) as mountpoints, simply to ensure better remote name resolution.

Now you could enter a command such as:

```
ls /net/qnet.qnx.com/repository
```

and you'd get a listing of the **repository** directory at the **qnet.qnx.com** site.

Chapter 14
TCP/IP Networking

As the Internet has grown to become more and more visible in our daily lives, the protocol it's based on—IP (Internet Protocol)—has become increasingly important. The IP protocol and tools that go with it are ubiquitous, making IP the de facto choice for many private networks.

IP is used for everything from simple tasks (e.g., remote login) to more complicated tasks (e.g., delivering realtime stock quotes). Most businesses are turning to the World Wide Web, which commonly rides on IP, for communication with their customers, advertising, and other business connectivity. The QNX Neutrino RTOS is well-suited for a variety of roles in this global network, from embedded devices connected to the Internet, to the routers that are used to implement the Internet itself.

Given these and many other user requirements, we've made our TCP/IP stack (included in `io-pkt*`) relatively light on resources, while using the common BSD API.

We provide the following stack configurations:

NetBSD TCP/IP stack

Based on the latest RFCs, including UDP, IP, and TCP. Also supports forwarding, broadcast and multicast, hardware checksum support, routing sockets, Unix domain sockets, multilink PPP, PPPoE, supernetting (CIDR), NAT/IP filtering, ARP, ICMP, and IGMP, as well as CIFS, DHCP, AutoIP, DNS, NFS (v2 and v3 server/client), NTP, RIP, RIPv2, and an embedded web server.

To create applications for this stack, you use the industry-standard BSD socket API. This stack also includes optimized forwarding code for additional performance and efficient packet routing when the stack is functioning as a network gateway.

Enhanced NetBSD stack with IPsec and IPv6

Includes all the features in the standard stack, plus the functionality targeted at the new generation of mobile and secure communications. This stack provides full IPv6 and IPsec (both IPv4 and IPv6) support through KAME extensions, as well as support for VPNs over IPsec tunnels.

This dual-mode stack supports IPv4 and IPv6 simultaneously and includes IPv6 support for autoconfiguration, which allows device configuration in plug-and-play network environments. IPv6 support includes IPv6-aware utilities and RIP/RIPng to support dynamic routing. An Advanced Socket API is also provided to supplement the standard socket API to take advantage of IPv6 extended-development capabilities.

IPsec support allows secure communication between hosts or networks, providing data confidentiality via strong encryption algorithms and data authentication features. IPsec support also includes the IKE (ISAKMP/Oakley) key management protocol for establishing secure host associations.

The QNX Neutrino TCP/IP suite is also modular. For example, it provides NFS as separate modules. With this kind of modularity, together with small-sized modules, embedded systems developers can more easily and quickly build small TCP/IP-capable systems.

Structure of the TCP/IP manager

As a resource manager, `io-pkt-*` benefits from the code savings and standard interface that all native resource managers enjoy. Due to the natural priority inheritance of QNX Neutrino IPC, clients will be dealt with in priority and time order, which leads to a more natural allocation of CPU resources.

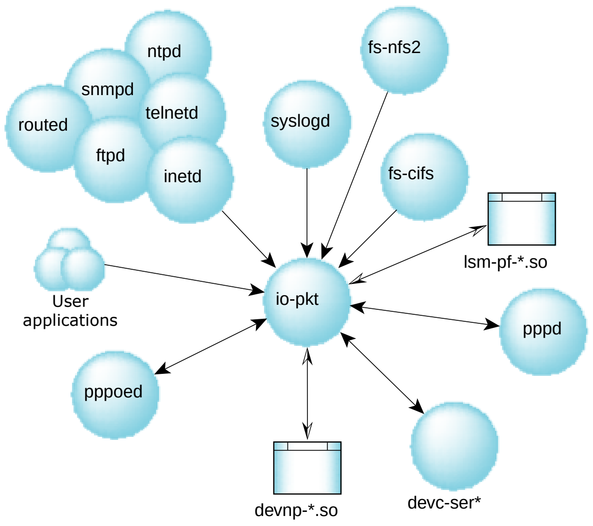


Figure 54: The `io-pkt` suite and its dependents.

PPP is implemented as part of `io-pkt*`. Since `io-pkt*` handles the transmission of PPP packets, there's no need for a memory copy of the packet data. This approach allows for high-performance PPPoE connections.

Other components of the TCP/IP suite (such as the NFS, etc.) are implemented outside of `io-pkt*`. This leads to better modularity and fault-tolerance.

Socket API

The BSD Socket API was the obvious choice for the QNX Neutrino RTOS. The Socket API is the standard API for TCP/IP programming in the UNIX world. In the Windows world, the Winsock API is based on and shares a lot with the BSD Socket API. This makes conversion between the two fairly easy.

All the routines that application programmers would expect are available, including (but not limited to):

- *accept()*
- *bind()*
- *bindresvport()*
- *connect()*
- *dn_comp()*
- *dn_expand()*
- *endprotoent()*
- *endservent()*
- *gethostbyaddr()*
- *gethostbyname()*
- *getpeername()*
- *getprotobyname()*
- *getprotobynumber()*
- *getprotoent()*
- *getservbyname()*
- *getservent()*
- *getsockname()*
- *getsockopt()*
- *herror()*
- *hstrerror()*
- *htonl()*
- *htons()*
- *h_errlist()*
- *h_errno()*
- *h_nerr()*
- *inet_addr()*
- *inet_aton()*
- *inet_lnaof()*
- *inet_makeaddr()*
- *inet_netof()*
- *inet_network()*
- *inet_ntoa()*

- *ioctl()*
- *listen()*
- *ntohl()*
- *ntohs()*
- *recv()*
- *recvfrom()*
- *res_init()*
- *res_mkquery()*
- *res_query()*
- *res_querydomain()*
- *res_search()*
- *res_send()*
- *select()*
- *send()*
- *sendto()*
- *setprotoent()*
- *setservent()*
- *setsockopt()*
- *shutdown()*
- *socket()*

For more information, see the QNX Neutrino *C Library Reference*.

The common daemons and utilities from the Internet will easily port or just compile in this environment. This makes it easy to leverage what already exists for your applications.

Database routines

The database routines listed below have been modified to better suit embedded systems.

/etc/resolv.conf

You can use configuration strings (via the *confstr()* function) to override the data usually contained in the **/etc/resolv.conf** file. You can also use the *RESCONF* environment variable to do this. Either method lets you use a nameserver without **/etc/resolv.conf**. This affects *gethostbyname()* and other resolver routines.

/etc/protocols

The *getprotobyname()* and *getprotobynumber()* functions have been modified to contain a small number of builtin protocols, including IP, ICNP, UDP, and TCP. For many applications, this means that the **/etc/protocols** file doesn't need to exist.

/etc/services

The *getservbyname()* function has been modified to contain a small number of builtin services, including `ftp`, `telnet`, `smtp`, `domain`, `nntp`, `netbios-ns`, `netbios-ssn`, `sunrpc`, and `nfsd`. For many applications, this means that the **/etc/services** file doesn't need to exist.

Multiple stacks

The QNX Neutrino network manager (`io-pkt`) lets you load *multiple* protocol shared objects. You can even run multiple, independent instances of the network manager (`io-pkt*`) itself. As with all QNX Neutrino system components, each `io-pkt*` naturally benefits from complete memory protection thanks to our microkernel architecture.

IP filtering and NAT

The IP filtering and NAT (Network Address Translation) `io-pkt*` module is a dynamically loadable TCP/IP stack module.

The **lsm-pf-*.so** module provides high-efficiency firewall services and includes such features as:

- rule grouping—to apply different groups of rules to different packets
- stateful filtering—an optional configuration to allow packets related to an already authorized connection to bypass the filter rules
- NAT—for mapping several internal addresses into a public (Internet) address, allowing several internal systems to share a single Internet IP address.
- proxy services—to allow `ftp`, `netbios`, and H.323 to use NAT
- port redirection—for redirecting incoming traffic to an internal server or to a pool of servers.

The IP filtering and NAT rules can be added or deleted *dynamically* to a running system. Logging services are also provided with the suite of utilities to monitor and control this module.

NTP

NTP (Network Time Protocol) allows you to keep the time of day for the devices in your network synchronized with the Internet standard time servers. The QNX Neutrino NTP daemon supports both server and client modes.

In server mode, a daemon on the local network synchronizes with the standard time servers. It will then broadcast or multicast what it learned to the clients on the local network, or wait for client requests. The client NTP systems will then be synchronized with the server NTP system. The NTP suite implements NTP v4 while maintaining compatibility with v3, v2, and v1.

Dynamic host configuration

We support DHCP (Dynamic Host Configuration Protocol), which is used to obtain TCP/IP configuration parameters.

Our DHCP client (`dhclient`) will obtain its configuration parameters from the DHCP server and configure the TCP/IP host for the user. This allows the user to add a host to the network without knowing what parameters (IP address, gateway, etc.) are required for the host. DHCP also allows a system administrator to control how hosts are added to the network. A DHCP server daemon (`dhcpd`) and relay agent (`dhcrelay`) are also provided to manage these clients.

For more information, see the entries for `dhclient`, `dhcrelay`, and `dhcpd` in the *Utilities Reference*.

AutoIP

Developed from the Zeroconf IETF draft, `autoipd` is an `io-pkt*` module that automatically configures the IPv4 address of your interface without the need of a server (as per DHCP) by negotiating with its peers on the network. This module can also coexist with DHCP (`dhclient`), allowing your interface to be assigned both a link-local IP address and a DHCP-assigned IP address at the same time.

PPP over Ethernet

We support the Point-to-Point Protocol over Ethernet (PPPoE), which is commonly deployed by broadband service providers.

Our PPPoE support consists of the `io-pkt-*` stack, as well as the `pppoectl` utility, which negotiates the PPPoE session. Once the PPPoE session is established, the `pppd` daemon creates a PPP connection.

When you use PPPoE, you don't need to specify any configuration parameters—our modules will automatically obtain the appropriate configuration data from your ISP and set up everything for you.

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

`io-pkt`

Networking manager.

`pppoectl`

Display or set parameters for a PPPoE interface.

/etc/autoconnect

Our autoconnect feature automatically sets up a connection to your ISP whenever a TCP/IP application is started. For example, suppose you want to start a dialup connection to the Internet. When your Web browser is started, it will pause and the **/etc/autoconnect** script will automatically dial your ISP. The browser will resume when the PPP session is established.

For more information, see the entry for **/etc/autoconnect** in the *Utilities Reference*.

Chapter 15
Cryptography Support

QNX Neutrino supports many cryptographic algorithms through features such as the OpenSSL library and utility and the QNX Neutrino `devcrypto` service.

OpenSSL version

OpenSSL cryptography support is available via the libraries **libcrypto.so.2** and **libssl.so.2** and the `openssl` utility.

QNX Neutrino includes the OpenSSL cryptography library version 1.0.2. For more information, including `openssl` commands, go to *https://www.openssl.org/docs/man1.0.2/*.

OpenSSL `devcrypto` extensions

The OpenSSL that QNX Neutrino provides allows you to redirect to `devcrypto` calls for cryptography operations that use the OpenSSL EVP API. This redirection allows access to EVP functions without recompiling application programs using OpenSSL.

For processes that use or link to the OpenSSL library (**libcrypto**), the following environment variable automatically redirects cryptography operations:

```
OPENSSL_CRYPTODEV=["alg1,alg2,..."|all] process_path args
```

where `"`*alg1*`,`*alg2*`,..."` are the algorithms that will be handled by `devcrypto` instead of OpenSSL's internal implementation. (Specify the algorithms using the names that OpenSSL uses internally.) Alternatively, specify `all` to redirect to `devcrypto` all algorithms that it supports; unsupported algorithms use OpenSSL.

In addition, specifying the *OPENSSL_CRYPTODEV_DEBUG* provides debug output related to the algorithm registration.

`devcrypto` service

The `devcrypto` service provides cryptography support through the "standard" **/dev/crypto** interface (similar to OpenBSD's cryptodev userspace API). The QNX Neutrino **/dev/crypto** is a driver that exposes an interface that uses I/O control calls to perform cryptography operations (MAC, digest, cipher, AEAD cipher, etc.). The `devcrypto` plugin API allows you to create a software backend to `devcrypto`, which provides access to either software and hardware cryptographic accelerators.

For more information, see the following documentation:

- The entry for **devcrypto** in the *Utilities Reference*
- “The `devcrypto` I/O command API (**cryptodev.h**)” in the *Security Developer's Guide*
- “The `devcrypto` plugin API (**devcrypto_plugin.h**)” in the *Security Developer's Guide*

Chapter 16
High Availability

The term *High Availability* (HA) is commonly used in telecommunications and other industries to describe a system's ability to remain up and running without interruption for extended periods of time.

The celebrated “five nines” availability metric refers to the percentage of uptime a system can sustain in a year—99.999% uptime amounts to about five minutes of downtime per year.

Obviously, an effective HA solution involves various hardware and software components that conspire to form a stable, working system. Assuming reliable hardware components with sufficient redundancy, how can an OS best remain stable and responsive when a particular component or application program fails? And in cases where redundant hardware may not be an option (e.g., consumer appliances), how can the OS itself support HA?

An OS for HA

If you had to design an HA-capable OS from the ground up, would you start with a single executable environment? In this simple, high-performance design, all OS components, device drivers, applications, the works, would all run without memory protection in kernel mode.

On second thought, maybe such an OS wouldn't be suited for HA, simply because if a single software component were to fail, the entire system would crash. And if you wanted to add a software component or otherwise modify the HA system, you'd have to take the system out of service to do so. In other words, the *conventional realtime executive* architecture wasn't built with HA in mind.

Suppose, then, that you base your HA-enabled OS on a separation of kernel space and user space, so that all applications would run in user mode and enjoy memory protection. You'd even be able to upgrade an application without incurring any downtime.

So far so good, but what would happen if a device driver, filesystem manager, or other essential OS component were to crash? Or what if you needed to add a new driver to a live system? You'd have to rebuild and restart the kernel. Based on such a *monolithic kernel* architecture, your HA system wouldn't be as available as it should be.

Inherent HA

A true microkernel that provides full memory protection is inherently the most stable OS architecture.

Very little code is running in kernel mode that could cause the kernel itself to fail. And individual processes, whether applications or OS services, can be started and stopped dynamically, without jeopardizing system uptime.

QNX Neutrino inherently provides several key features that are well-suited for HA systems:

- System stability through full memory protection for all OS and user processes.
- Dynamic loading and unloading of system components (device drivers, filesystem managers, etc.).
- Separation of all software components for simpler development and maintenance.

While any claims regarding “five nines” availability on the part of an OS must be viewed only in the context of the entire hardware/software HA system, one can always ask whether an OS truly has the appropriate underlying architecture capable of supporting HA.

HA-specific modules

Apart from its inherently robust architecture, QNX Neutrino also provides several components to help developers simplify the task of building and maintaining effective HA systems:

- *HA client-side library*—cover functions that allow for automatic and transparent recovery mechanisms for failed server connections.
- *HA Manager*—a “smart watchdog” that can perform multistage recovery whenever system services or processes fail.

Custom hardware support

While many operating systems provide HA support in a hardware-specific way (e.g., via PCI Hot Plug), QNX Neutrino isn't tied to PCI. Your particular HA system may be built on a custom chassis, in which case an OS that offers a PCI-based HA “solution” may not address your needs at all.

QNX Software Systems is an actively contributing member of the Service Availability Forum (*www.saforum.org*), an industry body dedicated to developing open, industry-standard specifications for building HA systems.

Client library

The High Availability client-side library provides a drop-in enhancement solution for many standard C Library I/O operations.

The HA library's cover functions allow for *automatic and transparent recovery mechanisms* for failed connections that can be recovered from in an HA scenario. Note that the HA library is both thread-safe and cancellation-safe.

The main principle of the client library is to provide drop-in replacements for all the message-delivery functions (i.e., *MsgSend**). A client can select which particular connections it would like to make highly available, thereby allowing all other connections to operate as ordinary connections (i.e., in a non-HA environment).

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 (e.g., EBADF). But in an HA-aware scenario, these transient faults are recovered from almost immediately, thus making the services available again.

Recovery example

The following example demonstrates a simple recovery scenario, where a client opens a file across a network filesystem.

If the NFS server were to die, the HA Manager would restart it and remount the filesystem. Normally, any clients that previously had files open across the old connection would now have a stale connection handle. But if the client uses the *ha_attach* functions, it can recover from the lost connection.

The *ha_attach* functions allow the client to provide a custom recovery function that's automatically invoked by the cover-function library. This recovery function could simply reopen the connection (thereby getting a connection to the new server), or it could perform a more complex recovery (e.g., adjusting the file position offsets and reconstructing its state with respect to the connection). This mechanism thus lets you develop arbitrarily complex recovery scenarios, while the cover-function library takes care of the details (detecting a failure, invoking recovery functions, and retransmitting state information).

```
#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 TESTFILE "/net/machine99/home/test/testfile"

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

```
int recover_conn(int oldfd, void *hdl)
{
  int newfd;
  Handle *thdl;
  thdl = (Handle *)hdl;
  newfd = ha_reopen(oldfd, TESTFILE, O_RDONLY);
  if (newfd >= 0) {
    // adjust file offset to previously known point
    lseek(newfd, thdl->curr_offset, SEEK_SET);
    // increment our count of successful recoveries
    (thdl->nr)++;
  }
  return(newfd);
}

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

  hdl.nr = 0;
  hdl.curr_offset = 0;
  // open a connection
  // recovery will be using "recovery_conn", and "hdl" will
  // be passed to it as a parameter
  fd = ha_open(TESTFILE, O_RDONLY, recover_conn, (void *)&hdl, 0);
  if (fd < 0) {
    printf("could not open file\n");
    exit(-1);
  }
  status = read(fd,buf,15);
  if (status < 0) {
    printf("error: %s\n",strerror(errno));
    exit(-1);
  }
  else {
    hdl.curr_offset += status;
  }
  fd2 = ha_dup(fd);
  // fs-nfs3 fails, and is restarted, the network mounts
  // are reinstated at this point.
  // Our previous "fd" to the file is stale
  sleep(18);
  // reading from dup-ped fd
  // will fail, and will recover via recover_conn
  status = read(fd,buf,15);
  if (status < 0) {
    printf("error: %s\n",strerror(errno));
    exit(-1);
  }
```

```
    else {
      hdl.curr_offset += status;
    }
    printf("total recoveries, %d\n",hdl.nr);
    ha_close(fd);
    ha_close(fd2);
    exit(0);
  }
```

Since the cover-function library takes over the lowest *MsgSend*()* calls, most standard library functions (*read()*, *write()*, *printf()*, *scanf()*, etc.) are also automatically HA-aware. The library also provides an *ha-dup()* function, which is semantically equivalent to the standard *dup()* function in the context of HA-aware connections. You can replace recovery functions *during the lifetime of a connection*, which greatly simplifies the task of developing highly customized recovery mechanisms.

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 whenever system services or processes fail, no longer respond, or are detected to be in a state where they cease to provide acceptable levels 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 itself into the native networking mechanism (Qnet), this framework transparently extends a local monitoring mechanism to a network-distributed one.

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. Again, the system could be simply 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 allows external detectors to detect and report interesting events to the system, and can associate actions with the occurrence of those events.

In many HA systems, each single point of failure (SPOF) 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.

HAM and the Guardian

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 the HAM is to explicitly instruct it to terminate the Guardian and then to terminate itself.

HAM hierarchy

The High Availability Manager consists of three main components.

- *Entities*
- *Conditions*
- *Actions*

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.

There are three fundamental entity types:

- *Self-attached* entities (HA-aware components)—processes that choose to send heartbeats to the HAM, which will then monitor them 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. In other words, this is a situation where a process says, "Do the following if I die."
- *Externally attached* entities (HA-unaware components)—generic processes (including legacy components) in the system that are being monitored. These could be arbitrary daemons/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 place holder for matching any entity. The global entity 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, and allows a process to say things like, "When *any* process dies or misses a heartbeat, do the following." The global entity is never added or removed, but only referred to. Conditions can be added to or removed from the global entity, of course, and actions can be added to or removed from any of the conditions.

Conditions

Conditions are associated with entities; a condition represents the entity's state.

Condition	Description
CONDDEATH	The entity has died.
CONDABNORMALDEATH	The entity has died an abnormal death. Whenever an entity dies, this condition is triggered by a mechanism that results in the generation of a core dump file.
CONDDETACH	The entity that was being monitored is detaching. This ends the HAM's monitoring of that entity.
CONDATTACH	An entity for whom a place holder was previously created (i.e., some process has subscribed to events relating to this entity) has joined the system. This is also the start of the HAM's monitoring of the entity.
CONDHBEATMISSEDHIGH	The entity missed sending a "heartbeat" message specified for a condition of "high" severity.

Condition	Description
CONDHBEATMISSEDLOW	The entity missed sending a "heartbeat" message specified for a condition of "low"
CONDRESTART	The entity was restarted. This condition is true *after* the entity is successfully restarted.
CONDRAISE	An externally detected condition is reported to the HAM. Subscribers can associate actions with these externally detected conditions.
CONDSTATE	An entity reports a state transition to the 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.

For the conditions listed above (except CONDSTATE, CONDRAISE, and CONDANY), the HAM is the publisher—it automatically detects and/or triggers the conditions. For the CONDSTATE and CONDRAISE conditions, external detectors publish the conditions to the HAM.

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 subscribers can selectively associate actions with individual conditions based on the information published.

Any condition can be associated as a wild card with any entity, so a process can associate actions with *any* condition in a specific entity, or even in any entity. Note that conditions are also associated with symbolic names, which also need to be unique within an entity.

Actions

Actions are associated with conditions. Actions are executed when the appropriate conditions are true with respect to a specific entity.

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

Action	Description
ham_action_restart()	This action restarts the entity
ham_action_execute()	Executes an arbitrary command (e.g., to start a process)
ham_action_notify_pulse()	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.
ham_action_notify_signal()	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.

Action	Description
ham_action_notify_pulse_node()	This is the same as *ham_action_notify_pulse()* above, except that the node name specified for the recipient of the pulse can be the fully qualified node name.
ham_action_notify_signal_node()	This is the same as *ham_action_notify_signal()* above, except that the node name specified for the recipient of the signal can be the fully qualified node name.
ham_action_waitfor()	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()	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()	Reports this condition to a logging mechanism.

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

What happens if an action itself fails? You can specify an *alternate list of actions* to be performed to recover from that failure. These alternate actions are associated with the primary actions through several *ham_action_fail** functions:

- *ham_action_fail_execute()*
- *ham_action_fail_notify_pulse()*
- *ham_action_fail_notify_signal()*
- *ham_action_fail_notify_pulse_node()*
- *ham_action_fail_notify_signal_node()*
- *ham_action_fail_waitfor()*
- *ham_action_fail_log()*

Publishing autonomously detected conditions

Entities or other components in the system can inform the HAM about conditions (events) that they deem interesting, and the HAM in turn can deliver these conditions (events) to other components in the system that have expressed interest in (subscribed to) them.

This publishing feature allows arbitrary 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 and/or preventive action.

There are currently two different ways of publishing information to the HAM; both of these are designed to be general enough to permit clients to build more complex information exchange mechanisms:

- publishing state transitions
- publishing other conditions.

State transitions

An entity can report its state transitions to the HAM, which maintains every entity's current state (as reported by the entity). The HAM doesn't interpret the meaning of the state value itself, nor does it try to validate the state transitions, but it can generate events based on transitions from one state to another.

Components can publish transitions that they want the external world to know about. These states needn't necessarily represent a specific state the application uses internally for decision making.

To notify the HAM of a state transition, components can use the *ham_entity_condition_state()* function. Since the HAM is interested only in the *next* state in the transition, this is the only information that's 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 (see *below*).

Other conditions

In addition to the above, components on the system can also publish autonomously detected conditions by using the *ham_entity_condition_raise()* 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. As a result of this call, the HAM triggers a condition-raise event internally, which other components can subscribe to using the *ham_condition_raise()* API call (see *below*).

Subscribing to autonomously published conditions

To express their interest in events published by other components, subscribers can use the *ham_condition_state()* and *ham_condition_raise()* API calls.

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

Trigger based on state transition

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're interested in by specifying values for the *fromstate* and *tostate* parameters in the *ham_condition_state* API call.

Trigger based on specific published condition

To express interest in conditions raised by entities, subscribers can use the API call *ham_condition_raise()*, indicating as parameters to the call what sort of conditions they're interested in.

For more information, see the API reference documentation in the High Availability Framework *Developer's Guide*.

HAM as a "filesystem"

Effectively, 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.

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`).

The **/proc/ham** filesystem presents a lot of information about the current state of the system's entities. It also provides useful statistics on heartbeats, restarts, and deaths, giving you a snapshot in time of the system's various entities, conditions, and actions.

Multistage recovery

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. In most cases, recovery requires more than a single restart mechanism in order to properly restore the system's state to what it was before a failure.

For example, suppose you've started `fs-nfs3` (the NFS filesystem) and then mounted a few directories from multiple sources. You can instruct HAM to restart `fs-nfs3` upon failure, and also to remount the appropriate directories as required after restarting the NFS process.

As another example, suppose `io-pkt*` (the network I/O manager) were to die. We can tell 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).

HAM API

The basic mechanism to talk to HAM is to use its API. This API is implemented as a library that you can link against. The library is thread-safe as well as cancellation-safe.

To control exactly what/how you're monitoring, the HAM API provides a collection of functions, including:

Function	Description
ham_action_control()	Perform control operations on an action object.
ham_action_execute()	Add an execute action to a condition.
ham_action_fail_execute()	Add to an action an execute action that will be executed if the corresponding action fails.
ham_action_fail_log()	Insert a log message into the activity log.
ham_action_fail_notify_pulse()	Add to an action a notify pulse action that will be executed if the corresponding action fails.
ham_action_fail_notify_pulse_node()	Add to an action a node-specific notify pulse action that will be executed if the corresponding action fails.

Function	Description
ham_action_fail_notify_signal()	Add to an action a notify signal action that will be executed if the corresponding action fails.
ham_action_fail_notify_signal_node()	Add to an action a node-specific notify signal action that will be executed if the corresponding action fails.
ham_action_fail_waitfor()	Add to an action a waitfor action that will be executed if the corresponding action fails.
ham_action_handle()	Get a handle to an action in a condition in an entity.
ham_action_handle_node()	Get a handle to an action in a condition in an entity, using a nodename.
ham_action_handle_free()	Free a previously obtained handle to an action in a condition in an entity.
ham_action_heartbeat_healthy()	Reset a heartbeat's state to healthy.
ham_action_log()	Insert a log message into the activity log.
ham_action_notify_pulse()	Add a notify-pulse action to a condition.
ham_action_notify_pulse_node()	Add a notify-pulse action to a condition, using a nodename.
ham_action_notify_signal()	Add a notify-signal action to a condition.
ham_action_notify_signal_node()	Add a notify-signal action to a condition, using a nodename.
ham_action_remove()	Remove an action from a condition.
ham_action_restart()	Add a restart action to a condition.
ham_action_waitfor()	Add a waitfor action to a condition.
ham_attach()	Attach an entity.
ham_attach_node()	Attach an entity, using a nodename.
ham_attach_self()	Attach an application as a self-attached entity.
ham_condition()	Set up a condition to be triggered when a certain event occurs.
ham_condition_control()	Perform control operations on a condition object.
ham_condition_handle()	Get a handle to a condition in an entity.
ham_condition_handle_node()	Get a handle to a condition in an entity, using a nodename.
ham_condition_handle_free()	Free a previously obtained handle to a condition in an entity.

Function	Description
ham_condition_raise()	Attach a condition associated with a condition raise condition that's triggered by an entity raising a condition.
ham_condition_remove()	Remove a condition from an entity.
ham_condition_state()	Attach a condition associated with a state transition condition that's triggered by an entity reporting a state change.
ham_connect()	Connect to a HAM.
ham_connect_nd()	Connect to a remote HAM.
ham_connect_node()	Connect to a remote HAM, using a nodename.
ham_detach()	Detach an entity from a HAM.
ham_detach_name()	Detach an entity from a HAM, using an entity name.
ham_detach_name_node()	Detach an entity from a HAM, using an entity name and a nodename.
ham_detach_self()	Detach a self-attached entity from a HAM.
ham_disconnect()	Disconnect from a HAM.
ham_disconnect_nd()	Disconnect from a remote HAM.
ham_disconnect_node()	Disconnect from a remote HAM, using a nodename.
ham_entity()	Create entity placeholder objects in a HAM.
ham_entity_condition_raise()	Raise a condition.
ham_entity_condition_state()	Notify the HAM of a state transition.
ham_entity_control()	Perform control operations on an entity object in a HAM.
ham_entity_handle()	Get a handle to an entity.
ham_entity_handle_node()	Get a handle to an entity, using a nodename.
ham_entity_handle_free()	Free a previously obtained handle to an entity.
ham_entity_node()	Create entity placeholder objects in a HAM, using a nodename.
ham_heartbeat()	Send a heartbeat to a HAM.
ham_stop()	Stop a HAM.

Function	Description
ham_stop_nd()	Stop a remote HAM.
ham_stop_node()	Stop a remote HAM, using a nodename.
ham_verbose()	Modify the verbosity of a HAM.

Chapter 17 Adaptive Partitioning

In many computer systems, it's important to protect different applications or groups of applications from others. You don't want one application—whether defective or malicious—to corrupt another or prevent it from running.

To address this issue, some systems use virtual walls, called *partitions*, around a set of applications to ensure that each partition is given an engineered set of resources. The primary resource considered is CPU time, but can also include any shared resource, such as memory and file space (disk or flash).

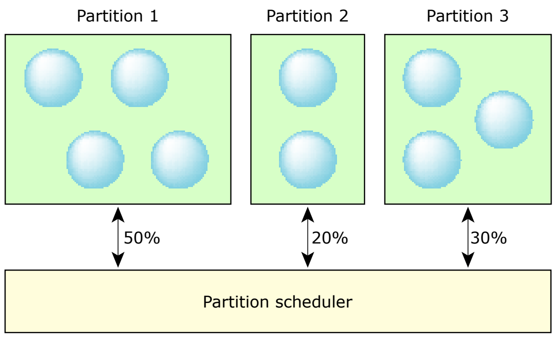


Figure 55: Static partitions guarantee that processes get the resources specified by the system designer.

Partitions provide:

- memory protection—each partition is discrete and controlled by the Memory Management Unit (MMU)
- overload protection—each partition is guaranteed a slice of execution time, as specified by the system designer

By using multiple partitions, you can avoid having a single point of failure. For example, a runaway process can't occupy the entire system's resources; processes in other partitions still receive their allocated share of system resources.

Even without adaptive partitioning, QNX Neutrino's process model provides significantly more protection than some other operating systems do, including:

- full memory protection between processes
- message-passing to provide uniform and controlled IPC
- priority inheritance with a clean client-server model
- hard realtime deterministic scheduling
- a detailed permission model for devices, files, and memory
- memory, file-descriptor, CPU, and priority limits, using the POSIX *setrlimit()* function to constrain runaway processes

Typically, the main objective of resource partitioning on other systems is to divide a computer into a set of smaller computers that interact as little as possible; however, this approach isn't very flexible. In QNX Neutrino, adaptive partitioning takes a much more flexible view.

Our partitions are adaptive because:

- you can change configurations at run time
- they're typically fixed at one configuration time
- the partition behavior auto-adapts to conditions at run time. For example:
  - free time is redistributed to other scheduler partitions
  - filesystems can bill time to clients with a mechanism that temporarily moves threads between time partitions

Why *adaptive*?

To provide realtime performance with guarantees against overloading, QNX Neutrino introduced *adaptive* partitioning. Rigid partitions work best in fairly static systems with little or no dynamic deployment of software. In dynamic systems, static partitions can be inefficient. For example, the static division of execution time between partitions can waste CPU time and introduce delays:

- If most of the partitions are idle, and one is very busy, the busy partition doesn't receive any additional execution time, while background threads in the other partitions waste CPU time.
- If an interrupt is scheduled for a partition, it has to wait until the partition runs. This can cause unacceptable latency, especially if bursts of interrupts occur.

An adaptive partition is a set of threads that work on a common or related goal or activity. Like a static partition, an adaptive partition has a budget allocated to it that guarantees its minimum share of the CPU's resources. Unlike a static partition, an adaptive partition:

- isn't locked to a fixed set of code in a static partition; you can dynamically add and configure adaptive partitions, as required
- behaves as a global hard realtime thread scheduler under normal load, but can still provide minimal interrupt latencies even under overload conditions
- maximizes the usage of the CPU's resources, by distributing a partition's unused budget among partitions that require extra resources when the system isn't loaded.

You can introduce adaptive partitioning without changing—or even recompiling—your application code, although you do have to rebuild your system's OS image.

You can have a maximum of 32 partitions. In QNX Neutrino, it's threads, not partitions, that are scheduled.

Benefits of adaptive partitioning

Adaptive partitioning provides a number of benefits to the design, development, running, and debugging of your system.

Engineering product performance

Adaptive partitioning lets you design your system so as to optimize its performance.

Partitions divide resources so that they can be used by a collection of programs. A partition represents a fraction of a resource and includes a few rules that define the resource usage. The resources include basic objects, such as processor cycles, program store or high-level objects, such as buffers, page tables, or file descriptors.

Adaptive partitioning ensures that any free time available in the system (i.e., CPU time in a partition's budget that the partition doesn't need) is made available to other partitions. This lets the system handle sudden processing demands that occur during normal system operation. With a cyclic thread scheduler, there's a “use it or lose it” approach where unused CPU time is spent running an idler thread in partitions that don't use their full budget.

Another important feature of adaptive partitioning is the concept of *partition inheritance*. This feature lets designers develop server processes that run with no (or minimal) budget. When the server performs requests from clients, the client partition is billed for the time. Without this feature, CPU budget would be allocated to a server regardless of how much or often it's used. The benefits of these features include:

- You don't have to over-engineer the system, so the overall cost decreases.
- If you add an application, you don't have to re-engineer the budget of common services, such as filesystems or servers.
- The system is faster and more responsive to the user.
- The system guarantees time for important tasks.
- You can use priorities to specify a process's urgency, and a partition's CPU budget to specify its importance.

Dealing with design complexity

Designing large-scale distributed systems is inherently complex. Typical systems have a large number of subsystems, processes, and threads developed in isolation from each other. The design is divided among groups with differing system performance goals, different schemes for determining priorities, and different approaches to runtime optimization.

This can be further compounded by product development in different geographic locations and time zones. Once all of these disparate subsystems are integrated into a common runtime environment, all parts of the system need to provide adequate response under all operating scenarios, such as:

- normal system loading
- peak periods
- failure conditions

Given the parallel development paths, system issues invariably arise when integrating the product. Typically, once a system is running, unforeseen interactions that cause serious performance degradations are uncovered. When situations such as this arise, there are usually very few designers or architects who can diagnose and solve these problems at a system level. Solutions often take considerable modifications (frequently, by trial and error) to get it right. This extends system integration, impacting the time to market.

Problems of this nature can take a week or more to troubleshoot, and several weeks to adjust priorities across the system, retest, and refine. If these problems can't be solved effectively, product scalability is limited.

This is largely due to the fact that there's no effective way to “budget” CPU use across these groups. Thread priorities provide a way to ensure that critical tasks run, but don't provide guaranteed CPU time for important, noncritical tasks, which can be starved in normal operations. In addition, a common approach to establishing thread priorities is difficult to scale across a large development team.

Adaptive partitioning using the thread scheduler lets architects maintain a reserve of resources for emergency purposes, such as a disaster-recovery system, or a field-debugging shell, and define high-level CPU budgets per subsystem, allowing development groups to implement their own priority schemes and optimizations within a given budget. This approach lets design groups develop subsystems independently and eases the integration effort. The net effect is to improve time-to-market and facilitate product scaling.

Providing security

Many systems are vulnerable to Denial of Service (DOS) attacks. For example, a malicious user could bombard a system with requests that need to be processed by one process. When under attack, this process overloads the CPU and effectively starves the rest of the system.

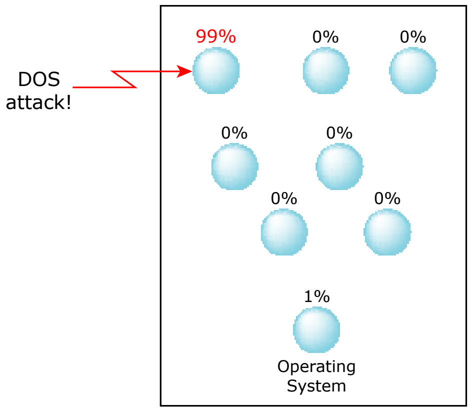


Figure 56: Without adaptive partitioning, a DOS attack on one process can starve other critical functions.

Some systems try to overcome this problem by implementing a monitor process that detects CPU utilization and invokes corrective actions when it deems that a process is using too much CPU. This approach has a number of drawbacks, including:

- Response time is typically slow.
- This approach caps the CPU usage in times when legitimate processing is required.
- It isn't infallible or reliable; it depends on appropriate thread priorities to ensure that the monitor process obtains sufficient CPU time.

Adaptive partitioning can solve this problem by providing separate budgets to the system's various functions. This ensures that the system always has some CPU capacity for important tasks. Threads can change their own priorities, which can be a security hole, but you can configure the thread scheduler to prevent code running in a partition from changing its own budget.

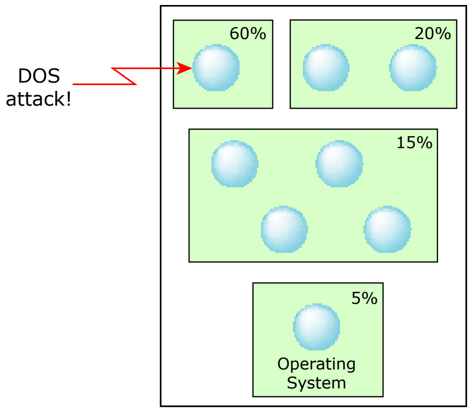


Figure 57: With scheduler partitions, a DOS attack is contained.

Since adaptive partitioning can allocate any unused CPU time to partitions that require it, it doesn't unnecessarily cap control-plane activity when there's a legitimate need for increased processing.

Debugging

Adaptive partitioning can even make debugging an embedded system easier—during development or deployment—by providing an “emergency door” into the system.

Simply create a partition that you can run diagnostic tools in; if you don't need to use the partition, the thread scheduler allocates its budget among the other partitions. This provides you with access to the system without compromising its performance. For more information, see the Testing and Debugging chapter of the Adaptive Partitioning *User's Guide*.

Adaptive partitioning thread scheduler

The thread scheduler is an optional scheduler that lets you guarantee minimum percentages of the CPU's throughput to groups of threads, processes, or applications. The percentage of the CPU time allotted to a partition is called a *budget*.

The thread scheduler has been designed on top of the core QNX Neutrino architecture primarily to solve these problems in embedded systems design:

- guaranteeing a specified minimum share of CPU time when the system is overloaded
- preventing unimportant or untrusted applications from monopolizing the system

For more information, see the Adaptive Partitioning *User's Guide*.

Appendix A
What is Real Time and Why Do I Need It?

Real time is an often misunderstood—and misapplied—property of operating systems. This appendix provides a summary of some of the critical elements of realtime computing and discusses a few design considerations and benefits.

We can start with a basic definition of a realtime system, as defined in the FAQ for the comp.realtime newsgroup:

> A realtime system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

Real time, then, is a property of systems where time is literally "of the essence." In a realtime system, the value of a computation depends on how timely the answer is. For example, a computation that's completed late has a diminishing value, or no value whatsoever, and a computation completed early is of no extra value. Real time is always a matter of degree, since even batch computing systems have a realtime aspect to them—nobody wants to get their payroll deposit two weeks late!

Problems arise when there's competition for resources in the system, and resources are shared among many activities, which is where we begin to apply the realtime property to operating systems. In implementing any realtime system, a critical step in the process is the determination of a schedule of activities such that all activities are completed on time.

Any realtime system comprises different types of activities:

- those that can be scheduled
- those that can't be scheduled, such as operating-system facilities and interrupt handlers
- non-realtime activities

If non-schedulable activities can execute in preference to schedulable activities, they'll affect the ability of the system to handle time constraints.

Hard and soft real time

Hard real time is a property of the timeliness of a computation in the system. A hard realtime constraint in the system is one for which there's no value to a computation if it's late, and the effects of a late computation may be catastrophic to the system. Simply put, a hard realtime system is one where all of the activities must be completed on time.

Soft real time is a property of the timeliness of a computation where the value diminishes according to its tardiness. A soft realtime system can tolerate some late answers to soft realtime computations, as long as the value hasn't diminished to zero. A soft realtime system often carries meta requirements, such as a stochastic model of acceptable frequency of late computations. Note that this is very different from conventional applications of the term, which don't account for how late a computation is completed or how frequently this may occur.

Soft real time is often improperly applied to operating systems that don't satisfy the necessary conditions for guaranteeing that computations can be completed on time. Such operating systems are best described as quasi-realtime or pseudo-realtime in that they execute realtime activities in preference to others whenever necessary, but don't adequately account for non-schedulable activities in the system.

Who needs real time?

Traditionally, realtime operating systems have been used in "mission-critical" environments requiring hard realtime capability, where failure to perform activities in a timely manner can result in harm to persons or property.

Often overlooked, however, are situations where there's a need to meet quality of service guarantees, particularly when failure to do so could result in financial penalty. This covers obvious service scenarios, such as "thirty minutes or it's free," but it also includes intangible penalties, such as lost opportunities or loss of market share.

More and more, real time is being employed in consumer devices—complex systems that demand the utmost in reliability. For example, a non-realtime device aimed at presenting live video, such as MPEG movies, that depends on software for any part of the delivery of the content, may experience dropped frames at a rate that the customer perceives as unacceptable.

In designing systems, developers need to assess whether the performance benefits warrant the use of realtime technology. A decision made early on can have unforeseen consequences when overload of the deployed system leads to pathological behavior in which most or none of the activities complete on time, if at all.

Realtime technology can be applied to conventional systems in ways that have a positive impact on the user experience, either by improving the perceived response to certain events, or ensuring that important activities execute preferentially with respect to others in the system.

What is a realtime OS?

Our definition of what constitutes a hard realtime operating system is based on realtime scheduling theory that's consistent with industry practice:

A hard realtime operating system must guarantee that a feasible schedule can be executed, given sufficient computational capacity if external factors are discounted. External factors, in this case, are devices that may generate interrupts, including network interfaces that generate interrupts in response to network traffic.

In other words, if a system designer controls the environment of the system, the operating system itself won't be the cause of any tardy computations. We can apply this term to conventional operating systems—which typically execute tasks according to their priority—by referring to scheduling theory and deriving a minimum set of conditions that must be met. Without getting into too much detail, scheduling theory demonstrates that a schedule can be translated into static priority assignments in a way that guarantees timeliness. It does so by dividing the time available into periodic divisions and assuming a certain proportion of each division is reserved for particular realtime activities.

In order to do so, the following basic requirements must be met:

1. Higher-priority tasks always execute in preference to lower-priority tasks.

2. Priority inversions, which may result when a higher-priority task needs a resource allocated to a lower-priority one, are bounded.
3. Non-schedulable activities, including both non-realtime activities and operating-system activities, don't exceed the remaining capacity in any particular division.

Because of condition 3, we must discount those activities outside of the control of the operating system, yielding the external factors provision above.

We can then derive the following operating system requirements (OSRs):

1. The OS must support fixed-priority preemptive scheduling for tasks (both threads and processes, as applicable).
2. The OS must provide priority inheritance or priority-ceiling emulation for synchronization primitives.
3. The OS kernel must be preemptible.
4. Interrupts must have a fixed upper bound on latency. By extension, nested interrupt support is required.
5. Operating-system services must execute at a priority determined by the client of the service:
   - All services on which it's dependent must inherit that priority.
   - Priority inversion avoidance must be applied to all shared resources used by the service.

OSR 3 and OSR 4 impose a fixed upper bound on the latency imposed on the onset of any particular realtime activity. OSR 5 ensures that operating system services themselves—which are internal factors—don't introduce non-schedulable activities into the system that could violate basic requirement 3.

How does an RTOS differ from a conventional OS?

The key characteristic that separates an RTOS from a conventional OS is the predictability that's inherent in all of the requirements above. A conventional OS, such as Linux, attempts to use a “fairness” policy in scheduling threads and processes to the CPU. This gives all applications in the system a chance to make progress, but doesn't establish the supremacy of realtime threads in the system or preserve their relative priorities, as is required to guarantee that they'll finish on time. Likewise, all priority information is usually lost when a system service, usually performed in a kernel call, is being performed on behalf of the client thread. This results in unpredictable delays preventing an activity from completing on time.

By contrast, the microkernel architecture used in the QNX Neutrino RTOS is designed to deal directly with all of these requirements.

The microkernel itself simply manages threads within the system and allows them to communicate with each other. Scheduling is always performed at the thread level, and threads are always scheduled according to their fixed priority—or, in the case of priority inversion, by the priority, as adjusted by the microkernel to compensate for priority inversions. A high-priority thread that becomes ready to run can preempt a lower-priority thread.

Within this framework, all device drivers and operating system services apart from basic scheduling and interprocess communication (IPC) exist as separate processes within the system. All services are accessed through a synchronous message-passing IPC mechanism that allows the receiver to inherit the priority of the client. This priority-inheritance scheme allows OSR 5 to be met by carrying the priority of the original realtime activity into all service requests and subsequent device-driver requests.

There's an attendant flexibility available as well. Since OSR 1 and OSR 5 stress that device-driver requests need to operate in priority order, at the priority of the client, throughput for normal operations can be substantially reduced. Using this model, an operating service or device driver can be swapped out in favor of a realtime version that satisfies these requirements. Complex systems generally partition such resources into realtime and non-realtime with different service and device-driver implementations for each resource.

Because of the above, all activities in the system are performed at a priority determined by the thread on whose behalf they're operating.

What is a soft realtime OS?

A soft realtime OS must be capable of doing effectively everything that a hard realtime OS must do. In addition, a soft realtime OS must be capable of providing monitoring capabilities with accurate cost accounting on the tasks in the system. It must determine when activities have failed to complete on time or when they have exceeded their allocated CPU capacity, and trigger the appropriate response.

How does all of this affect my application?

If you're writing an application or system for deployment on a realtime OS, it's important to consider the effect that the RTOS characteristics have on the execution of the application, and to understand how it can be used to your benefit. For example, with an RTOS you can increase responsiveness of certain operations initiated by the user.

Most applications normally run at the default user priority within the system. This means that applications normally run in a round robin execution, competing with each other for a proportion of the CPU capacity. Without the type of realtime schedule mentioned above, you can manipulate the priorities of the processes in the system to have certain activities run preferentially to others in the system. Manipulation of the priorities is a double-edged sword. Used judiciously, it can dramatically improve response in areas that are important to the user. At the same time, it's possible to starve other processes in the system in a way that typically doesn't happen on a conventional desktop system.

The key to ensuring that higher-priority processes and threads don't starve out other processes in the system is to be certain of the limits imposed on their execution. By pacing the execution, or by throttling it in response to load, you can limit the proportion of CPU consumed by these activities so user processes get their share of the CPU.

Media players, such as audio players (MP3, **.wav**, etc.) and video (MPEG-2), are a good example of applications that can benefit from priority manipulation. The operation of a media player can be tied to the media rate that's required for proper playback (i.e., 44 kHz audio, 30 fps video). So within this constraint, a reader thread that buffers data and a rendering or playback thread can both be designed to awaken on a programmable timer, buffer or render a single frame, and then go to sleep awaiting the next timer trigger. This provides the necessary pacing, so that the priority can be assigned above normal user activities, but below more critical system functions.

By choosing appropriate priorities, you can ensure that playback occurs consistently at the given media rate. A well-written media player also takes into account quality of service, so that if it doesn't receive adequate CPU time, it can reduce its requirements by selectively dropping samples or follow an appropriate fall-back strategy. This then prevents it from starving other processes as well.

You may also wish to treat certain user events preferentially within the system. This works well when you increase the concurrency within an application, and when the event can always be handled in a predictable, small amount of time. The key concern here is the frequency at which these events can be generated. If they can't occur too frequently, it's safe to raise the priority of the thread responding to them. If they can occur too frequently, other activities will be starved under overload conditions.

The simplest solution is to divide responsibility for events into different handling threads with different priorities and to queue requests or deliver them with messages. You can tie the handler's execution to a timer, so that the execution of the thread is throttled by the timer, handling a fixed number of requests within a given interval. This stresses the importance of factoring areas of application responsibility, giving a flexible design with opportunities for effective use of concurrency and preferential response, all of which lead to a greater feel of responsiveness.

Appendix B Glossary

adaptive

Scheduling policy whereby a thread's priority is decayed by 1. See also *FIFO*, *round robin*, and *sporadic*.

adaptive partitioning

A method of dividing, in a flexible manner, CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

application ID

A number that identifies all processes that are part of an application. Like process group IDs, the application ID value is the same as the process ID of the first process in the application. A new application is created by spawning with the POSIX_SPAWN_NEWAPP or SPAWN_NEWAPP flag. A process created without one of those inherits the application ID of its parent. A process needs the PROCMGR_AID_CHILD_NEWAPP ability in order to set those flags.

The *SignalKill()* kernel call accepts a SIG_APPID flag ORed into the signal number parameter. This tells it to send the signal to all the processes with an application ID that matches the *pid* argument. The DCMD_PROC_INFO *devctl()* returns the application ID in a structure field.

asymmetric multiprocessing (AMP)

A multiprocessing system where a separate OS, or a separate instantiation of the same OS, runs on each CPU.

atomic

Of or relating to atoms. :-)

In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered *indivisible*. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread's position.

attributes structure

Structure containing information used on a per-resource basis (as opposed to the *OCB*, which is used on a per-open basis).

This structure is also known as a *handle*. The structure definition is fixed (`iofunc_attr_t`), but may be extended. See also *mount structure*.

bank-switched

A term indicating that a certain memory component (usually the device holding an *image*) isn't entirely addressable by the processor. In this case, a hardware component manifests a small portion (or “window”) of the device onto the processor's address bus. Special commands have to be issued to the hardware to move the window to different locations in the device. See also *linearly mapped*.

base layer calls

Convenient set of library calls for writing resource managers. These calls all start with *resmgr_*()*. Note that while some base layer calls are unavoidable (e.g., *resmgr_attach()*), we recommend that you use the *POSIX layer calls* where possible.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an “extension” to the BIOS or ROM Monitor—control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes 0x55 and 0xAA must be present (in that order) as the first two bytes in the device, with control being transferred to offset 0x0003.

block-integral

The requirement that data be transferred such that individual structure components are transferred in their entirety—no partial structure component transfers are allowed.

In a resource manager, directory data must be returned to a client as *block-integral* data. This means that only complete `struct dirent` structures can be returned—it's inappropriate to return partial structures, assuming that the next _IO_READ request will “pick up” where the previous one left off.

bootable

An image can be either bootable or *nonbootable*. A bootable image is one that contains the startup code that the IPL can transfer control to.

bootfile

The part of an OS image that runs the *startup code* and the microkernel.

bound multiprocessing (BMP)

A multiprocessing system where a single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

budget

In *sporadic* scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for `mkifs` specifying the contents and other details of an *image*, or for `mkefs` specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or “cooked” mode. In this mode, the character device library performs line-editing operations on each received character. Only when a line is “completely entered”—typically when a carriage return (CR) is received—will the line of data be made available to application processes. Contrast *raw mode*.

channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a *connection* (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using *ChannelCreate()*), and then receives messages from that channel (using *MsgReceive()*). Another thread that wishes to send a message to the first thread must make a connection to that channel by “attaching” to the channel (using *ConnectAttach()*) and then sending data (using *MsgSend()*).

chid

An abbreviation for *channel ID*.

CIFS

Common Internet File System (also known as SMB)—a protocol that allows a client workstation to perform transparent file access over a network to a Windows 95/98/NT server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

CIS

Card Information Structure—a data block that maintains information about flash configuration. The CIS description includes the types of memory devices in the regions, the physical geometry of these devices, and the partitions located on the flash.

coid

An abbreviation for *connection ID*.

combine message

A resource manager message that consists of two or more messages. The messages are constructed as combine messages by the client's C library (e.g., *stat()*, *readblock()*), and then handled as individual messages by the resource manager.

The purpose of combine messages is to conserve network bandwidth and/or to provide support for atomic operations. See also *connect message* and *I/O message*.

connect message

In a resource manager, a message issued by the client to perform an operation based on a pathname (e.g., an `io_open` message). Depending on the type of connect message sent, a context block (see *OCB*) may be associated with the request and will be passed to subsequent I/O messages. See also *combine message* and *I/O message*.

connection

A kernel object used with message passing.

Connections are created by client threads to "connect" to the channels made available by servers. Once connections are established, clients can *MsgSendv()* messages over them. If a number of threads in a process all attach to the same channel, then the one connection is shared among all the threads. Channels and connections are identified within a process by a small integer.

The key thing to note is that connections and file descriptors (*FD*) are one and the same object. See also *channel* and *FD*.

context

Information retained between invocations of functionality.

When using a resource manager, the client sets up an association or *context* within the resource manager by issuing an *open()* call and getting back a file descriptor. The resource manager is responsible for storing the information required by the context (see *OCB*). When the client issues further file-descriptor based messages, the resource manager uses the OCB to determine the context for interpretation of the client's messages.

cooked mode

See *canonical mode*.

core dump

A file describing the state of a process that terminated abnormally.

critical section

A code passage that *must* be executed "serially" (i.e., by only one thread at a time). The simplest from of critical section enforcement is via a *mutex*.

deadlock

A condition in which one or more threads are unable to continue due to resource contention. A common form of deadlock can occur when one thread sends a message to another, while the other thread sends a message to the first. Both threads are now waiting for each other to reply to the message. Deadlock can be avoided by good design practices or massive kludges—we recommend the good design approach.

device driver

A process that allows the OS and application programs to use the underlying hardware in a generic way (e.g., a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for the QNX Neutrino RTOS are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn't affect any other part of the OS—drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won't cause the entire OS to shut down.

discrete (or traditional) multiprocessor system

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

DNS

Domain Name Service—an Internet protocol used to convert ASCII domain names into IP addresses. In QNX Neutrino native networking, `dns` is one of *Qnet*'s built-in resolvers.

dynamic bootfile

An OS image built on the fly. Contrast *static bootfile*.

dynamic linking

The process whereby you link your modules in such a way that the Process Manager will link them to the library modules before your program runs. The word "dynamic" here means that the association between your program and the library modules that it uses is done *at load time*, not at link time. Contrast *static linking*. See also *runtime loading*.

edge-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is "noticed" upon a transition to/from the rising/falling edge of a pulse. Contrast *level-sensitive*.

edited mode

See *canonical mode*.

EOI

End Of Interrupt—a command that the OS sends to the PIC after processing all Interrupt Service Routines (ISR) for that particular interrupt source so that the PIC can reset the processor's In Service Register. See also *PIC* and *ISR*.

EPROM

Erasable Programmable Read-Only Memory—a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g., 12 V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. Changing a bit from a 0 state into a 1 state can be accomplished only by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically 10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast *flash* and *RAM*.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requestor of the event.

FD

File Descriptor—a client must open a file descriptor to a resource manager via the *open()* function call. The file descriptor then serves as a handle for the client to use in subsequent messages. Note that a file descriptor is the exact same object as a connection ID (*coid*, returned by *ConnectAttach()*).

FIFO

First In First Out—a scheduling policy whereby a thread is able to consume CPU at its priority level without bounds. See also *adaptive*, *round robin*, and *sporadic*.

flash memory

A memory technology similar in characteristics to *EPROM* memory, with the exception that erasing is performed electrically instead of via ultraviolet light, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64 KB at a time) instead of the entire device. Contrast *EPROM* and *RAM*.

FQNN

Fully Qualified Node Name—a unique name that identifies a QNX Neutrino node on a network. The FQNN consists of the nodename plus the node domain tacked together.

garbage collection

Also known as space reclamation, the process whereby a filesystem manager recovers the space occupied by deleted files and directories.

HA

High Availability—in telecommunications and other industries, HA describes a system's ability to remain up and running without interruption for extended periods of time.

handle

A pointer that the resource manager base library binds to the pathname registered via *resmgr_attach()*. This handle is typically used to associate some kind of per-device information. Note that if you use the *iofunc_*() POSIX layer calls*, you must use a particular *type* of handle—in this case called an *attributes structure*.

hard thread affinity

A user-specified binding of a thread to a set of processors, done by means of a *runmask*. Contrast *soft thread affinity*.

image

In the context of embedded QNX Neutrino systems, an “image” can mean either a structure that contains files (i.e., an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim FFS-2-compatible filesystem (i.e., a flash filesystem image).

inherit mask

A bitmask that specifies which processors a thread's children can run on. Contrast *runmask*.

interrupt

An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.

interrupt handler

See *ISR*.

interrupt latency

The amount of elapsed time between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service routine. Also designated as "T_{il}". Contrast *scheduling latency*.

interrupt service routine

See *ISR*.

interrupt service thread

A thread that is responsible for performing thread-level servicing of an interrupt.

Since an *ISR* can call only a very limited number of functions, and since the amount of time spent in an ISR should be kept to a minimum, generally the bulk of the interrupt servicing work should be done by a thread. The thread attaches the interrupt (via *InterruptAttach()* or *InterruptAttachEvent()*) and then blocks (via *InterruptWait()*), waiting for the ISR to tell it to do something (by returning an event of type SIGEV_INTR). To aid in minimizing *scheduling latency*, the interrupt service thread should raise its priority appropriately.

I/O message

A message that relies on an existing binding between the client and the resource manager. For example, an _IO_READ message depends on the client's having previously established an association (or *context*) with the resource manager by issuing an *open()* and getting back a file descriptor. See also *connect message*, *context*, *combine message*, and *message*.

I/O privileges

Particular rights, that, if enabled for a given thread, allow the thread to perform I/O instructions (such as the x86 assembler `in` and `out` instructions). By default, I/O privileges are disabled, because a program with them enabled can wreak havoc on a system. To enable I/O privileges, the process must have the PROCMGR_AID_IO ability enabled (see *procmgr_ability()*), and the thread must call *ThreadCtl()*.

IPC

Interprocess Communication—the ability for two processes (or threads) to communicate. The QNX Neutrino RTOS offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), as well as signals.

IPL

Initial Program Loader—the software component that either takes control at the processor's reset vector (e.g., location 0xFFFFFFF0 on the x86), or is a BIOS extension. This component is responsible for setting up the machine into a usable state, such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also *BIOS extension signature* and *startup code*.

IRQ

Interrupt Request—a hardware request line asserted by a peripheral to indicate that it requires servicing by software. The IRQ is handled by the *PIC*, which then interrupts the processor, usually causing the processor to execute an *Interrupt Service Routine (ISR)*.

ISR

Interrupt Service Routine—a routine responsible for servicing hardware (e.g., reading and/or writing some device ports), for updating some data structures shared between the ISR and the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.

kernel

See *microkernel*.

level-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast *edge-sensitive*.

linearly mapped

A term indicating that a certain memory component is entirely addressable by the processor. Contrast *bank-switched*.

message

A parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message—the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various “changes of state” that affect when, and for how long, they may run.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mount structure

An optional, well-defined data structure (of type `iofunc_mount_t`) within an *iofunc_*()* structure, which contains information used on a per-mountpoint basis (generally used only for filesystem resource managers). See also *attributes structure* and *OCB*.

mountpoint

The location in the pathname space where a resource manager has "registered" itself. For example, the serial port resource manager registers mountpoints for each serial device (**/dev/ser1**, **/dev/ser2**, etc.), and a CD-ROM filesystem may register a single mountpoint of **/cdrom**.

multicore system

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (*pthread_mutex_lock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a *critical section*). See also *critical section*.

name resolution

In a QNX Neutrino network, the process by which the *Qnet* network manager converts an *FQNN* to a list of destination addresses that the transport layer knows how to get to.

name resolver

Program code that attempts to convert an *FQNN* to a destination address.

nd

An abbreviation for *node descriptor*, a numerical identifier for a node *relative to the current node*. Each node's node descriptor for itself is 0 (ND_LOCAL_NODE).

NDP

Node Discovery Protocol—proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.

network directory

A directory in the pathname space that's implemented by the *Qnet* network manager.

NFS

Network FileSystem—a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local pathname space. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g., `ls`, `cp`, `mv`) operate on the remote files exactly as they do on your local files.

NMI

Nonmaskable Interrupt—an interrupt that can't be masked by the processor. We don't recommend using an NMI!

Node Discovery Protocol

See *NDP*.

node domain

A character string that the *Qnet* network manager tacks onto the nodename to form an *FQNN*.

nodename

A unique name consisting of a character string that identifies a node on a network.

nonbootable

A nonbootable OS image is usually provided for larger embedded systems or for small embedded systems where a separate, configuration-dependent setup may be required. Think of it as a second “filesystem” that has some additional files on it. Since it's nonbootable, it typically won't contain the OS, startup file, etc. Contrast *bootable*.

OCB

Open Control Block (or Open Context Block)—a block of data established by a resource manager during its handling of the client's *open()* function. This context block is bound by the resource manager to this particular request, and is then automatically passed to all subsequent I/O functions generated by the client on the file descriptor returned by the client's *open()*.

partition

A division of CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

pathname prefix

See *mountpoint*.

pathname space mapping

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

persistent

When applied to storage media, the ability for the medium to retain information across a power-cycle. For example, a hard disk is a persistent storage medium, whereas a ramdisk is not, because the data is lost when power is lost.

PIC

Programmable Interrupt Controller—hardware component that handles IRQs. See also *edge-sensitive*, *level-sensitive*, and *ISR*.

PID

Process ID. Also often *pid* (e.g., as an argument in a function call).

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface—the “X” alludes to “UNIX”, on which the interface is based.

POSIX layer calls

Convenient set of library calls for writing resource managers. The POSIX layer calls can handle even more of the common-case messages and functions than the *base layer calls*. These calls are identified by the *iofunc_*()* prefix. In order to use these (and we strongly recommend that you do), you must also use the well-defined POSIX-layer *attributes* (`iofunc_attr_t`), *OCB* (`iofunc_ocb_t`), and (optionally) *mount* (`iofunc_mount_t`) structures.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been “preempted” by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY, the lower-priority thread is immediately preempted by the higher-priority thread.

prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent *priority inversion*.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU *on behalf of* the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it's working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one *thread* running in it—this thread is then called the first thread.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.

process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of `1`.

pty

Pseudo-TTY—a character-based device that has two “ends”: a master end and a slave end. Data written to the master end shows up on the slave end, and vice versa. These devices are typically used to interface between a program that expects a character device and another program that wishes to use that device (e.g., the shell and the `telnet` daemon process, used for logging in to a system over the Internet).

pulses

In addition to the synchronous Send/Receive/Reply services, QNX Neutrino also supports fixed-size, nonblocking messages known as pulses. These carry a small payload (four bytes of data plus a single byte code). A pulse is also one form of *event* that can be returned from an ISR or a timer. See *MsgDeliverEvent()* for more information.

Qnet

The native network manager in the QNX Neutrino RTOS.

QoS

Quality of Service—a policy (e.g., `loadbalance`) used to connect nodes in a network in order to ensure highly dependable transmission. QoS is an issue that often arises in high-availability (*HA*) networks as well as realtime control systems.

RAM

Random Access Memory—a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast *flash* and *EPROM*.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data—you don't want any translations of the raw binary stream between the device and the application. Contrast *canonical mode*.

replenishment

In *sporadic* scheduling, the period of time during which a thread is allowed to consume its execution *budget*.

reset vector

The address at which the processor begins executing instructions after the processor's reset line has been activated. On the x86, for example, this is the address 0xFFFFFFF0.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX Neutrino resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g., serial ports, parallel ports, network cards, disk drives) or virtual (e.g., **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers*. But unlike device drivers, QNX Neutrino resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also *device driver*.

RMA

Rate Monotonic Analysis—a set of methods used to specify, analyze, and predict the timing behavior of realtime systems.

round robin

A scheduling policy whereby a thread is given a certain period of time to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also *adaptive*, *FIFO*, and *sporadic*.

runmask

A bitmask that indicates which processors a thread can run on. Contrast *inherit mask*.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast *static linking*.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as "T_{sl}". Contrast *interrupt latency*.

scoid

An abbreviation for *server connection ID*.

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

soft thread affinity

The scheme whereby the microkernel tries to dispatch a thread to the processor where it last ran, in an attempt to reduce thread migration from one processor to another, which can affect cache performance. Contrast *hard thread affinity*.

software interrupts

Similar to a hardware interrupt (see *interrupt*), except that the source of the interrupt is software.

sporadic

A scheduling policy whereby a thread's priority can oscillate dynamically between a “foreground” or normal priority and a “background” or low priority. A thread is given an execution *budget* of time to be consumed within a certain *replenishment* period. See also *adaptive*, *FIFO*, and *round robin*.

startup code

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

static bootfile

An image created at one time and then transmitted whenever a node boots. Contrast *dynamic bootfile*.

static linking

The process whereby you combine your modules with the modules from the library to form a single executable that's entirely self-contained. The word “static” implies that it's not going to change—*all* the required modules are already combined into one.

symmetric multiprocessing (SMP)

A multiprocessor system where a single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under the QNX Neutrino RTOS. A thread is a flow of execution; it exists within the context of a *process*.

tid

An abbreviation for *thread ID*.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an *event*, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a *round-robin* or *adaptive* scheduled thread. This period of time is small (on the order of tens of milliseconds); the actual value shouldn't be relied upon by any program (it's considered bad design).

TLB

An abbreviation for *translation look-aside buffer*. To maintain performance, the processor caches frequently used portions of the external memory page tables in the TLB.

TLS

An abbreviation for *thread local storage*.

Index