



QNX® NEUTRINO® RTOS V6.3

SYSTEM ARCHITECTURE



QNX[®] Neutrino[®] RTOS

System Architecture

For release 6.3.0 or later

Printed under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: <http://www.qnx.com/>

© 1996 – 2005, QNX Software Systems. All rights reserved.

Publishing history

October 1996	First edition
May 1999	Second edition
August 2001	Third edition
January 2003	Fourth edition
June 2004	Fifth edition

Electronic edition published 2005

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (www.qnx.com). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

QNX, Momentics, Neutrino, and Photon microGUI are registered trademarks of QNX Software Systems in certain jurisdictions. All other trademarks and trade names belong to their respective owners.

Printed in Canada.

Part Number: 002502

Contents

About This Guide xv

Typographical conventions	xvii
Note to Windows users	xviii
What you'll find in this guide	xix
Note to Windows users	xix

1 The Philosophy of QNX Neutrino 1

Design goals	3
An embeddable POSIX OS?	3
Product scaling	4
Why POSIX for embedded systems?	4
Why QNX Neutrino for embedded systems?	6
Microkernel architecture	7
The OS as a team of processes	10
A true kernel	11
System processes	12
Interprocess communication	13
QNX Neutrino as a message-passing operating system	13
Network distribution of kernels	14
Single-computer model	15
Flexible networking	15

2 The QNX Neutrino Microkernel 17

Introduction	19
The implementation of QNX Neutrino	20

POSIX realtime and thread extensions	20
System services	20
Threads and processes	22
Thread attributes	26
Thread scheduling	31
When scheduling decisions are made	31
Scheduling priority	32
Scheduling algorithms	34
IPC issues	41
Thread complexity issues	42
Synchronization services	44
Mutual exclusion locks	45
Condition variables	46
Barriers	48
Sleepon locks	51
Reader/writer locks	51
Semaphores	52
Synchronization via scheduling algorithm	53
Synchronization via message passing	54
Synchronization via atomic operations	54
Synchronization services implementation	55
QNX Neutrino IPC	56
Synchronous message passing	57
Message copying	58
Simple messages	62
Channels and connections	62
Message-passing API	66
Robust implementations with Send/Receive/Reply	67
Events	69
Signals	72
Summary of signals	77
POSIX message queues	79

Shared memory	81
Pipes and FIFOs	89
Clock and timer services	90
Time correction	92
Timers	93
Interrupt handling	95
Interrupt latency	96
Scheduling latency	97
Nested interrupts	98
Interrupt calls	98

3 The Instrumented Microkernel 105

Introduction	107
Instrumentation at a glance	107
Event control	108
Modes of emission	109
Ring buffer	110
Data interpretation	110
System analysis with the IDE	111
Proactive tracing	112

4 SMP 115

Introduction	117
SMP versions of procnto*	117
Booting an x86 SMP system	118
Booting a PowerPC or MIPS SMP system	119
How the SMP microkernel works	119
Scheduling	120
Hard processor affinity	120
Kernel locking	120
Inter-processor interrupts (IPIs)	121
Critical sections	121

5 Process Manager 125

- Introduction 127
- Process management 127
 - Process primitives 128
 - Process loading 133
- Memory management 133
 - Memory Management Units (MMUs) 134
 - Memory protection at run time 136
 - Quality control 138
 - Full-protection model 139
- Pathname management 140
 - Domains of authority 140
 - Resolving pathnames 141
 - Symbolic prefixes 145
 - File descriptor namespace 148

6 Dynamic Linking 153

- Shared objects 155
 - Statically linked 155
 - Dynamically linked 155
 - Augmenting code at runtime 156
- How shared objects are used 157
 - ELF format 157
 - ELF without COFF 158
 - The process 158
 - Runtime linker 160
 - Loading a shared library at runtime 161
 - Symbol name resolution 162

7 Resource Managers 165

- Introduction 167
- What is a resource manager? 167
 - Why write a resource manager? 168

The types of resource managers	170
Communication via native IPC	171
Resource manager architecture	173
Message types	174
The resource manager shared library	174
Summary	180

8 Filesystems 181

Introduction	183
Filesystems and pathname resolution	183
Filesystem classes	184
Filesystems as shared libraries	185
io-blk	187
Filesystem limitations	190
Image filesystem	191
RAM “filesystem”	192
Embedded transaction filesystem (ETFS)	193
Inside a transaction	194
Types of storage media	195
Reliability features	195
QNX4 filesystem	198
DOS Filesystem	199
CD-ROM filesystem	202
FFS3 filesystem	202
Customization	203
Organization	203
Features	205
Utilities	207
System calls	208
NFS filesystem	208
CIFS filesystem	209
Linux Ext2 filesystem	209
Virtual filesystems	209

Package filesystem 210

Inflator 211

9 Character I/O 213

Introduction 215

Driver/**io-char** communication 216

Device control 218

Input modes 219

Device subsystem performance 223

Console devices 224

Terminal emulation 225

Serial devices 225

Parallel devices 225

Pseudo terminal devices (ptys) 226

10 Networking Architecture 227

Introduction 229

Network manager (**io-net**) 230

Filter module 231

Converter module 231

Protocol module 231

Driver module 232

Loading and unloading a driver 232

Network DDK 233

11 Native Networking (Qnet) 235

QNX Neutrino distributed 237

Name resolution and lookup 239

File descriptor (connection ID) 241

Behind a simple *open()* 241

Global Name Service (GNS) 243

Network naming 243

Quality of Service (QoS) and multiple paths 245

QoS policies	245
Specifying QoS policies	248
Symbolic links	249
Examples	249
Local networks	250
Remote networks	250
Custom device drivers	251

12 TCP/IP Networking 253

Introduction	255
Stack configurations	255
Structure of TCP/IP manager	257
Socket API	258
Database routines	259
Multiple stacks	260
SCTP	260
IP filtering and NAT	261
NTP	262
Dynamic host configuration	262
AutoIP	262
PPP over Ethernet	263
/etc/autoconnect	263
SNMP support	264
Embedded web server	264
CGI method	265
SSI method	265

13 High Availability 267

What is High Availability?	269
An OS for HA	269
Custom hardware support	271
Client library	271
Recovery example	271

High Availability Manager	274
HAM and the Guardian	275
HAM hierarchy	275
Publishing autonomously detected conditions	280
Subscribing to autonomously published conditions	282
HAM as a “filesystem”	282
Multistage recovery	283
HAM API	283

14 Power Management 289

Power management for embedded systems	291
Application-driven framework	291
Framework components	291
Libraries and BSPs	292
Power manager	294
Power-managed objects	299
Power-aware applications	303
Persistent storage services	304
CPU power management	306

15 The Photon microGUI 309

A graphical microkernel	311
The Photon event space	312
Regions	314
Events	315
Graphics drivers	317
Multiple graphics drivers	318
Color model	319
Font support	319
Stroke-based fonts	320
Unicode multilingual support	320
UTF-8 encoding	321
Animation support	321

Multimedia support	322
Plugin architecture	322
Media Player	323
Printing support	324
The Photon Window Manager	324
Widget library	325
Fundamental widgets	325
Container widgets	329
Advanced widgets	332
Convenience functions	338
Driver development kits	342
Summary	342

Glossary 345

Index 369



List of Figures

Conventional executives offer no memory protection.	9
In a monolithic OS, system processes have no protection.	9
A microkernel provides complete memory protection.	9
The QNX Neutrino architecture.	10
The QNX Neutrino microkernel.	19
QNX Neutrino preemption details.	22
Sparse matrix (tid, key) to value mapping.	27
Possible thread states.	29
The ready queue.	33
Thread A blocks, Thread B runs.	35
FIFO scheduling.	36
Round-robin scheduling.	36
A thread's budget is replenished periodically.	38
Thread drops in priority until its budget is replenished.	39
Thread oscillates between high and low priority.	40
State changes in a send-receive-reply transaction.	57
A multipart transfer.	59
Scatter/gather of a read of 1454 bytes.	60
Connections map elegantly into file descriptors.	64
Threads blocked while in a channel queue.	65
Pulses pack a small payload.	66
Threads should always send up to higher-level threads.	68
A higher-level thread can “send” a pulse event.	69
The client sends a sigevent to the server.	71
Signal delivery.	74
Arguments to <i>mmap()</i> .	86

Interrupt handler simply terminates.	96
Interrupt handler terminates, returning an event.	97
Stacked interrupts.	98
Instrumentation at a glance.	108
The IDE helps you visualize system activity.	112
Virtual address mapping (on an x86).	135
Full protection VM (on an x86).	140
The SCOID and FD map to an OCB of an I/O Manager.	149
Two processes open the same file.	150
A process opens a file twice.	151
Object file format: linking view and execution view.	158
Process memory layout on an x86.	159
A resource manager is responsible for three data structures.	178
Multiple clients opening various devices.	179
Encapsulation.	180
QNX Neutrino filesystem layering.	186
ETFS is a filesystem composed entirely of transactions.	194
The io-char module is implemented as a library.	216
Device I/O in QNX Neutrino.	217
Conditions for satisfying an input request.	220
Pseudo-ttys.	226
The io-net process.	230
A simple network where the client and server reside on separate machines.	239
A client-server message pass across the network.	241
Qnet and a single network.	246
Qnet and physically separate networks.	247
The npm-tcpip suite and dependents.	257
Embedded web server.	264
The QNX PM framework facilitates application-specific designs.	294
Photon regions.	313
Opaque regions are clipped out.	317

About This 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	<code>if(stream == NULL)</code>
Command options	<code>-lR</code>
Commands	<code>make</code>
Environment variables	<code>PATH</code>
File and pathnames	<code>/dev/null</code>
Function names	<code>exit()</code>
Keyboard chords	Ctrl – Alt – Delete
Keyboard input	<code>something you type</code>
Keyboard keys	Enter
Program output	<code>login:</code>
Programming constants	<code>NULL</code>
Programming data types	<code>unsigned short</code>
Programming literals	<code>0xFF, "message string"</code>
Variable names	<code>stdin</code>
User-interface components	<code>Cancel</code>

We format single-step instructions like this:

- To reload the current page, press Ctrl – R.

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.



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

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

What you'll find in this guide

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

The 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, the Photon microGUI, and other aspects of QNX Neutrino.



Note that certain features of the OS as described in this guide may still be under development for a given release.

For the latest news and information on any QNX product, visit our website (www.qnx.com). You'll find links to many useful areas — software downloads, featured articles by developers, newsgroups, technical support options, and more.

Note to Windows users

In QNX documentation, we use a forward slash (/) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.



Chapter 1

The Philosophy of QNX Neutrino

In this chapter...

Design goals	3
Why QNX Neutrino for embedded systems?	6
Microkernel architecture	7
Interprocess communication	13
Network distribution of kernels	14



Design goals

The primary goal of QNX Neutrino 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, ARM, XScale, PowerPC, MIPS, and SH-4.

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 tradeoffs 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 applications than 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 inhouse 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 an up-to-date status of the many POSIX drafts/standards documents, see the PASC (Portable Applications Standards Committee of the IEEE Computer Society) report at <http://pasc.opengroup.org/standing/sd11.html>.

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: native and cross development

With the addition of interface hardware similar to the target runtime system, a workstation running a POSIX OS can become a functional superset of the embedded system. As a result, the application can be conveniently developed on the self-hosted desktop system.

Even in a cross-hosted development environment, the API remains essentially the same. Regardless of the particular host (QNX Neutrino, Linux, Windows,...) or the target (x86, ARM, MIPS, PowerPC,...), 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.

QNX Neutrino 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 a 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.

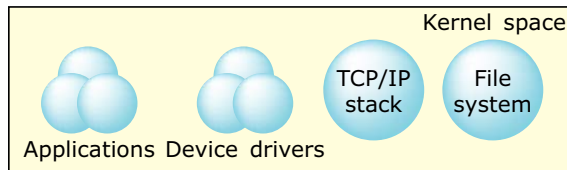
The real goal in designing a microkernel OS is not simply to “make it small.” A microkernel OS embodies a fundamental change in the approach to delivering OS functionality. *Modularity is the key, size is but a side effect.* To call any kernel a “microkernel” simply because it happens to be small would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

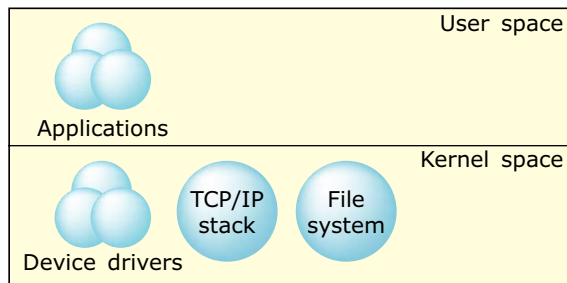
The microkernel differs from an executive in how the IPC services are used to extend the functionality of the kernel with additional, service-providing processes. Since the OS is implemented as a team of cooperating processes managed by the microkernel, user-written processes can serve both as applications 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.

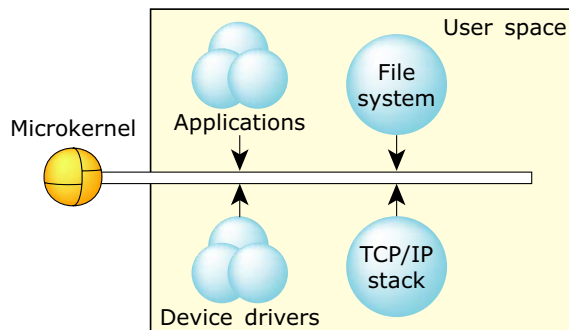
As the following diagrams show, a true microkernel offers *complete memory protection*, not only for user applications, but also for OS components (device drivers, filesystems, etc.):



Conventional executives offer no memory protection.



In a monolithic OS, system processes have no protection.

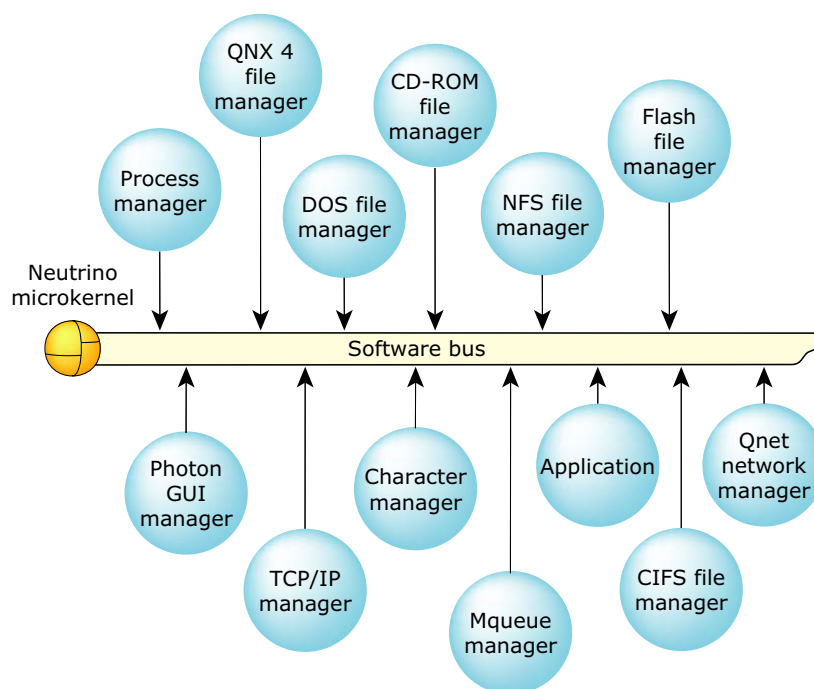


A microkernel provides complete memory protection.

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: QNX Neutrino, our most capable, scalable OS to date. We believe that this time-tested experience is what enables the QNX Neutrino OS to deliver the functionality it does using the limited resources it consumes.

The OS as a team of processes

The QNX Neutrino OS 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.



The QNX Neutrino 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 algorithms.
- **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
- graphical user interface (Photon)
- 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 QNX Neutrino 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.

QNX Neutrino as a message-passing operating system

QNX 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

In this chapter...

Introduction	19
The implementation of QNX Neutrino	20
System services	20
Threads and processes	22
Thread scheduling	31
Synchronization services	44
QNX Neutrino IPC	56
Clock and timer services	90
Interrupt handling	95

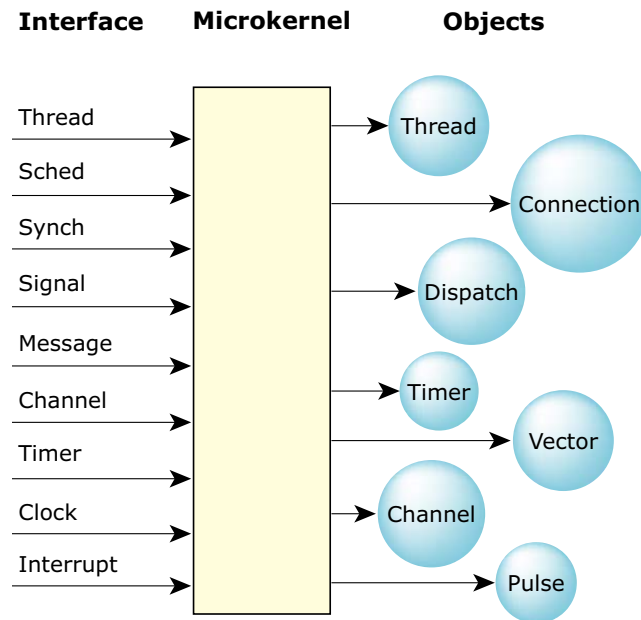


Introduction

The QNX Neutrino 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 microkernel (file and device I/O, for example) are provided by optional processes and shared libraries.

Successive QNX microkernels 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 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.



The QNX Neutrino 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 QNX Neutrino

Historically, the “application pressure” on QNX 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 Multi-Processing) 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 QNX Neutrino 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 QNX Neutrino 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.

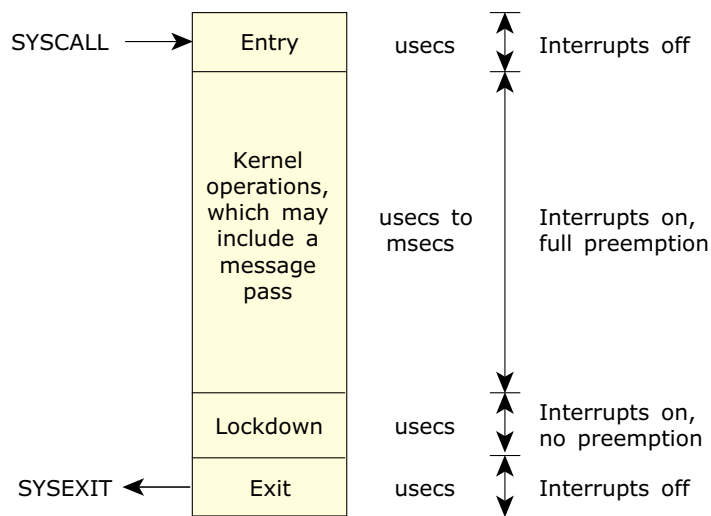
The entire OS is built upon these calls. The OS is fully preemptable, even while passing messages between processes; it resumes the message pass where it left off before preemption.

The minimal complexity of the microkernel helps place an upper bound on the longest nonpreemptable 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).



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, QNX Neutrino 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_testcancel()
```

pthread_key_create()
pthread_key_delete()
pthread_once()
pthread_self()
pthread_setcancelstate()
pthread_setcanceltype()

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
<i>pthread_create()</i>	<i>ThreadCreate()</i>	Create a new thread of execution.
<i>pthread_exit()</i>	<i>ThreadDestroy()</i>	Destroy a thread.
<i>pthread_detach()</i>	<i>ThreadDetach()</i>	Detach a thread so it doesn't need to be joined.
<i>pthread_join()</i>	<i>ThreadJoin()</i>	Join a thread waiting for its exit status.
<i>pthread_cancel()</i>	<i>ThreadCancel()</i>	Cancel a thread at the next cancellation point.
N/A	<i>ThreadCtl()</i>	Change a thread's Neutrino-specific thread characteristics.
<i>pthread_mutex_init()</i>	<i>SyncTypeCreate()</i>	Create a mutex.
<i>pthread_mutex_destroy()</i>	<i>SyncDestroy()</i>	Destroy a mutex.
<i>pthread_mutex_lock()</i>	<i>SyncMutexLock()</i>	Lock a mutex.
<i>pthread_mutex_trylock()</i>	<i>SyncMutexLock()</i>	Conditionally lock a mutex.
<i>pthread_mutex_unlock()</i>	<i>SyncMutexUnlock()</i>	Unlock a mutex.

continued...

POSIX call	Microkernel call	Description
<i>pthread_cond_init()</i>	<i>SyncTypeCreate()</i>	Create a condition variable.
<i>pthread_cond_destroy()</i>	<i>SyncDestroy()</i>	Destroy a condition variable.
<i>pthread_cond_wait()</i>	<i>SyncCondvarWait()</i>	Wait on a condition variable.
<i>pthread_cond_signal()</i>	<i>SyncCondvarSignal()</i>	Signal a condition variable.
<i>pthread_cond_broadcast()</i>	<i>SyncCondvarSignal()</i>	Broadcast a condition variable.
<i>pthread_getschedparam()</i>	<i>SchedGet()</i>	Get scheduling parameters and policy of thread.
<i>pthread_setschedparam()</i>	<i>SchedSet()</i>	Set scheduling parameters and policy of thread.
<i>pthread_sigmask()</i>	<i>SignalProcMask()</i>	Examine or set a thread's signal mask.
<i>pthread_kill()</i>	<i>SignalKill()</i>	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.

Thread attributes

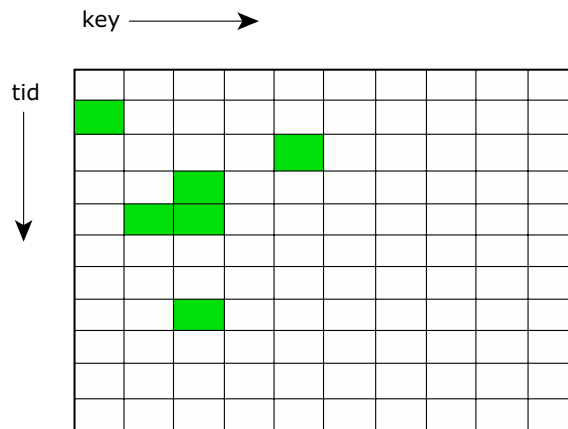
Although threads within a process share everything within the process's address space, each thread still has some "private" data. In some cases, this private data is protected within the kernel (e.g. the *tid* or thread ID), while other private data resides unprotected in the process's address space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

<i>tid</i>	Each thread is identified by an integer thread ID, starting at 1. The <i>tid</i> is unique within the thread's process.
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	<p>A thread has a system-defined data area called "thread local storage" (TLS). The TLS is used to store "per-thread" information (such as <i>tid</i>, <i>pid</i>, stack base, <i>errno</i>, 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.</p>
cancellation handlers	<p>Callback functions that are executed when the thread terminates.</p>

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.



*Sparse matrix (**tid**, **key**) to value mapping.*

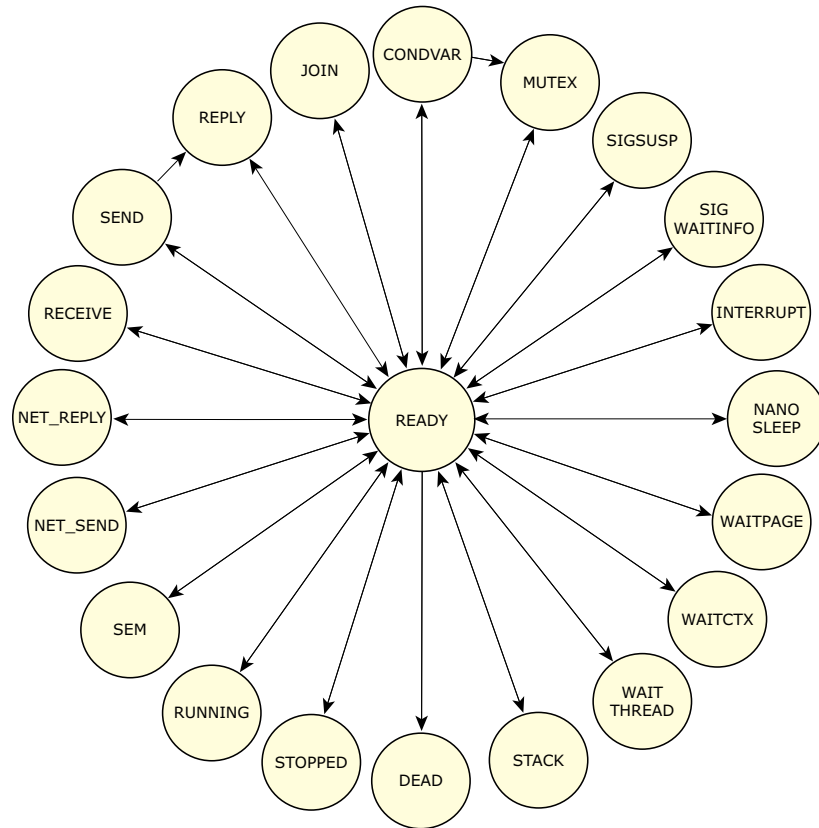
You use the following functions to create and manipulate this data:

Function	Description
<i>pthread_key_create()</i>	Create a data key with destructor function
<i>pthread_key_delete()</i>	Destroy a data key
<i>pthread_setspecific()</i>	Bind a data value to a data key
<i>pthread_getspecific()</i>	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:



Possible thread states.

CONDVAR	The thread is blocked on a condition variable (e.g. it called <i>pthread_condvar_wait()</i>).
DEAD	The thread has terminated and is waiting for a join by another thread.
INTERRUPT	The thread is blocked waiting for an interrupt (i.e. it called <i>InterruptWait()</i>).
JOIN	The thread is blocked waiting to join another thread (e.g. it called <i>pthread_join()</i>).

MUTEX	The thread is blocked on a mutual exclusion lock (e.g. it called <i>pthread_mutex_lock()</i>).
NANOSLEEP	The thread is sleeping for a short time interval (e.g. it called <i>nanosleep()</i>).
NET_REPLY	The thread is waiting for a reply to be delivered across the network (i.e. it called <i>MsgReply*()</i>).
NET_SEND	The thread is waiting for a pulse or signal to be delivered across the network (i.e. it called <i>MsgSendPulse()</i> , <i>MsgDeliverEvent()</i> , or <i>SignalKill()</i>).
READY	The thread is waiting to be executed while the processor executes another thread of equal or higher priority.
RECEIVE	The thread is blocked on a message receive (e.g. it called <i>MsgReceive()</i>).
REPLY	The thread is blocked on a message reply (i.e. it called <i>MsgSend()</i> , and the server received the message).
RUNNING	The thread is being executed by a processor.
SEM	The thread is waiting for a semaphore to be posted (i.e. it called <i>SyncSemWait()</i>).
SEND	The thread is blocked on a message send (e.g. it called <i>MsgSend()</i> , but the server hasn't yet received the message).
SIGSUSPEND	The thread is blocked waiting for a signal (i.e. it called <i>sigsuspend()</i>).
SIGWAITINFO	The thread is blocked waiting for a signal (i.e. it called <i>sigwaitinfo()</i>).

STACK	The thread is waiting for the virtual address space to be allocated for the thread's stack (parent will have called <i>ThreadCreate()</i>).
STOPPED	The thread is blocked waiting for a SIGCONT signal.
WAITCTX	The thread is waiting for a noninteger (e.g. floating point) context to become available for use.
WAITPAGE	The thread is waiting for physical memory to be allocated for a virtual address.
WAITTHREAD	The thread is waiting for a child thread to finish creating itself (i.e. it called <i>ThreadCreate()</i>).

Thread scheduling

When scheduling decisions are made

The execution of a running thread is temporarily suspended whenever the microkernel is 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 execution of the suspended thread will resume, but the scheduler will perform a context switch from one thread to another whenever the running thread:

- is blocked
- is preempted
- yields.

When thread is blocked

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 ready queue and the highest-priority ready thread is then run. When the blocked thread is subsequently unblocked, it is placed on the end of the ready queue for that priority level.

When thread is 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 remains at the beginning of the ready queue for that priority and the higher-priority thread runs.

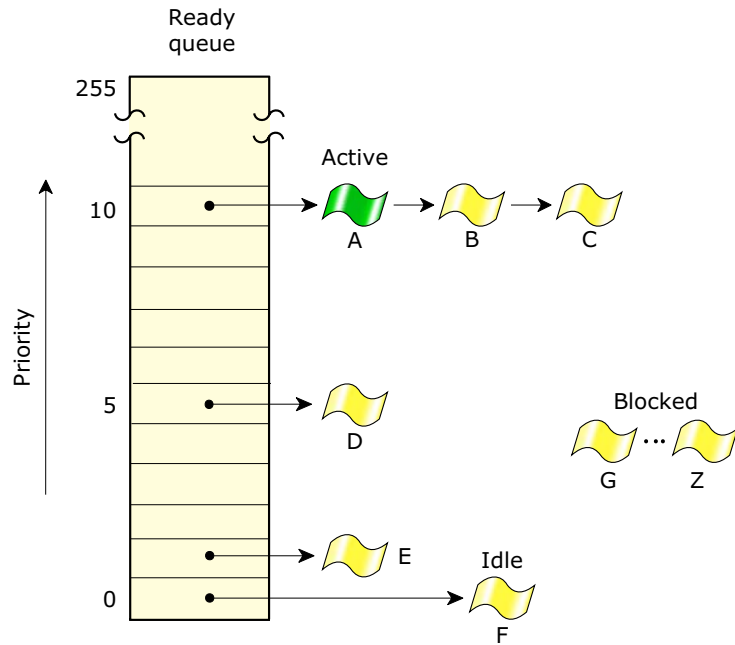
When thread yields

The running thread voluntarily yields the processor (*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 scheduler selects the next thread to run by looking at the priority assigned to every thread that is READY (i.e. capable of using the CPU). The thread with the highest priority is selected to run.

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



The ready queue.

The OS supports a total of 256 scheduling priority levels. A non-**root** thread can set its priority to a level from 1 to 63 (the highest priority), *independent of the scheduling policy*. Only **root** threads (i.e. those whose effective *uid* is 0) 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 non-**root** processes with the **procnto -P** option:

procnto -P *priority*

Here's a summary of the ranges:

Priority level	Owner
0	Idle thread
1 through <i>priority</i> – 1	Non- root or root
<i>priority</i> through 255	root

Note that in order to prevent *priority inversion*, the kernel may temporarily boost a thread's priority.

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

Scheduling algorithms

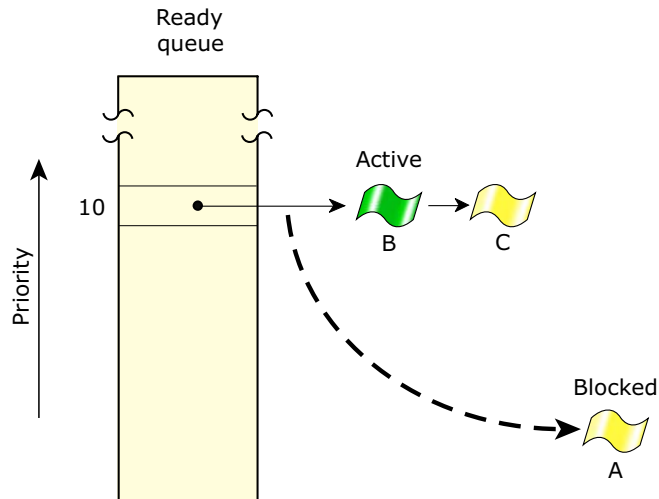
To meet the needs of various applications, QNX Neutrino provides these scheduling algorithms:

- 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 algorithms 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.



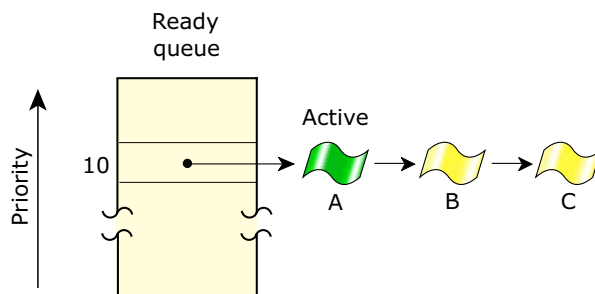
Thread A blocks, Thread B runs.

Although a thread inherits its scheduling algorithm 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.



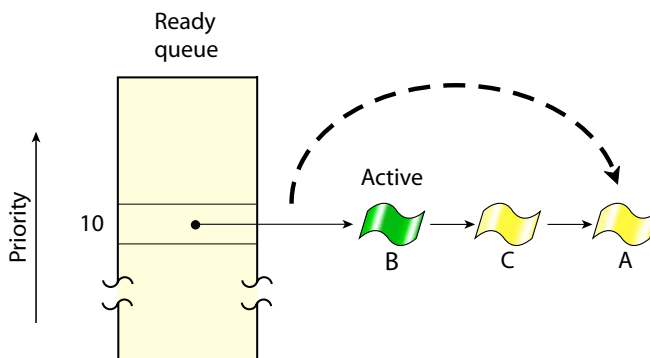
FIFO scheduling.

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:



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 $4 \times$ the clock period. (For more information, see the entry for *ClockPeriod()* in the *Library Reference*.)



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

Sporadic scheduling

The sporadic scheduling algorithm 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.

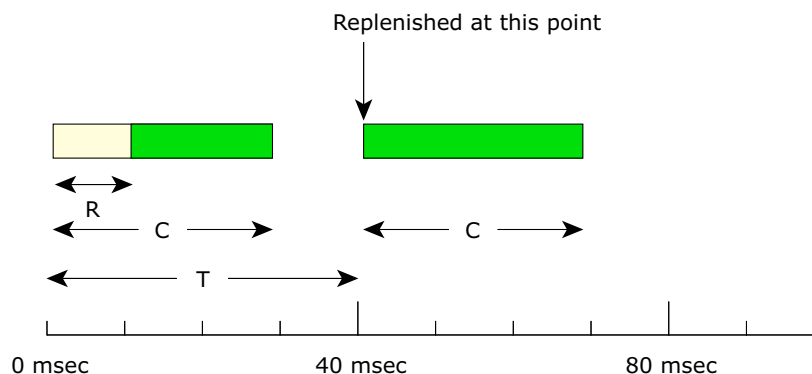
Max number of pending replenishments

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.

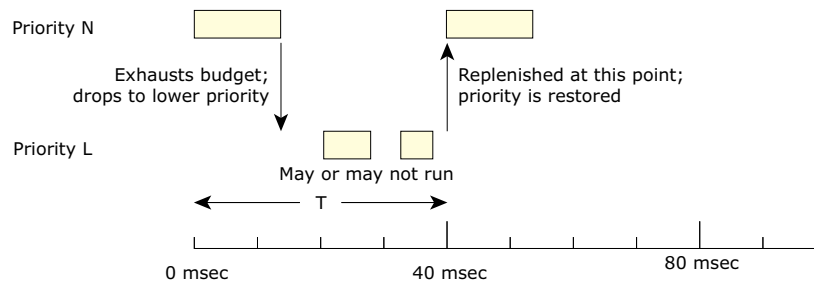
As the following diagram shows, the sporadic scheduling policy establishes a thread's initial execution budget (C), which is consumed by the thread as it runs and is replenished periodically (for the amount T). When a thread blocks, the amount of the execution budget that's been consumed (R) is arranged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



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:

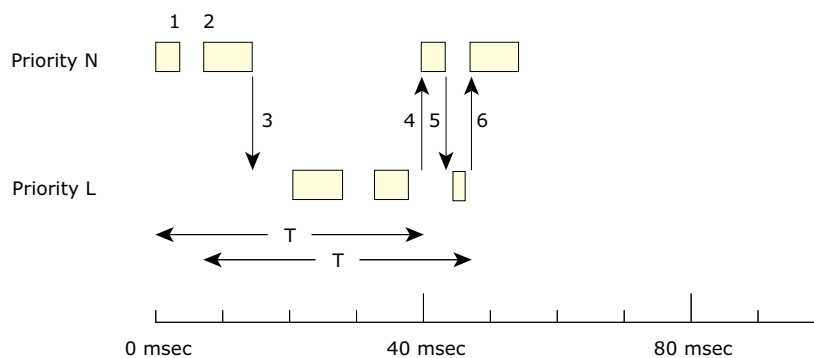


Thread drops in priority until its budget is replenished.

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 percent of the system's resources.

When a thread blocks multiple times, then several replenishment operations may be started and occur at different times. This could mean that the thread's execution budget will total C within a period T; however, the execution budget may not be contiguous during that period.



Thread oscillates between high and low priority.

In the diagram above, the thread has a budget (C) of 10 msec to be consumed within each 40-msec replenishment period (T).

- 1** The initial run of the thread is blocked after 3 msec, so a replenishment operation of 3 msec is scheduled to begin at the 40-msec mark, i.e. when its first replenishment period has elapsed.
- 2** The thread gets an opportunity to run again at 6 msec, which marks the start of another replenishment period (T). The thread still has 7 msec remaining in its budget.
- 3** The thread runs without blocking for 7 msec, thereby exhausting its budget, and then drops to low priority L, where it may or may not be able to execute. A replenishment of 7 msec is scheduled to occur at 46 msec ($40 + 6$), i.e. when T has elapsed.
- 4** The thread has 3 msec of its budget replenished at 40 msec (see Step 1) and is therefore boosted back to its normal priority.
- 5** The thread consumes the 3 msec of its budget and then is dropped back to the low priority.
- 6** The thread has 7 msec of its budget replenished at 46 msec (see Step 3) and is boosted back to its normal priority.

And so on. The thread will continue to oscillate between its two priority levels, servicing aperiodic events in your system in a controlled, predictable manner.

Manipulating priority and scheduling algorithms

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. Here are the POSIX calls for performing these manipulations, along with the microkernel calls used by these library routines:

POSIX call	Microkernel call	Description
<i>sched_getparam()</i>	<i>SchedGet()</i>	Get scheduling priority
<i>sched_setparam()</i>	<i>SchedSet()</i>	Set scheduling priority
<i>sched_getscheduler()</i>	<i>SchedGet()</i>	Get scheduling policy
<i>sched_setscheduler()</i>	<i>SchedSet()</i>	Set scheduling policy

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. QNX Neutrino’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

QNX Neutrino 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 LAN
Mutexes	Yes	No
Condvars	Yes	No
Barriers	No	No
Sleepon locks	No	No
Reader/writer locks	Yes	No
Semaphores	Yes	Yes (named only)
FIFO scheduling	Yes	No
Send/Receive/Reply	Yes	Yes
Atomic operations	Yes	No



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

- barriers, sleep-on 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).
-

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. It is typically acquired (*pthread_mutex_lock()*) 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 is later unlocked. 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.

On most processors, 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 done at acquisition time only 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 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

If a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner will be increased to that of the higher-priority blocked thread waiting for the mutex. The owner will return to its real priority when it unlocks the mutex. 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 attributes of the mutex can also be modified (using *pthread_mutex_setrecursive()*) 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.



Recursive mutexes are *non-POSIX* services — they don't work with 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. multi-processor 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.

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 <time.h>
#include <pthread.h>
#include <sys/neutrino.h>

pthread_barrier_t barrier; // barrier synchronization object

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));
}

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));
}

```

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
<i>pthread_barrierattr_getpshared()</i>	Get the value of a barrier's process-shared attribute
<i>pthread_barrierattr_destroy()</i>	Destroy a barrier's attributes object
<i>pthread_barrierattr_init()</i>	Initialize a barrier's attributes object
<i>pthread_barrierattr_setpshared()</i>	Set the value of a barrier's process-shared attribute
<i>pthread_barrier_destroy()</i>	Destroy a barrier
<i>pthread_barrier_init()</i>	Initialize a barrier
<i>pthread_barrier_wait()</i>	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.

Semaphores

Semaphores are another common form of synchronization that allows threads to “post” (*sem_post()*) and “wait” (*sem_wait()*) on a semaphore to control when threads wake or sleep. The post operation increments the semaphore; the 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.

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 uses a resource manager and as such allows semaphores to be used 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 wakeup, correct usage requires a loop:

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

Synchronization via scheduling algorithm

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



CAUTION: This exclusive-access relationship doesn’t apply in multi-processor 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).

Under QNX Neutrino, we provide 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 reenale 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
<i>SyncTypeCreate()</i>	<i>pthread_mutex_init()</i> , <i>pthread_cond_init()</i> , <i>sem_init()</i>	Create object for mutex, condvars, and semaphore.
<i>SyncDestroy()</i>	<i>pthread_mutex_destroy()</i> , <i>pthread_cond_destroy()</i> , <i>sem_destroy()</i>	Destroy synchronization object.
<i>SyncCondvarWait()</i>	<i>pthread_cond_wait()</i> , <i>pthread_cond_timedwait()</i>	Block on a condvar.
<i>SyncCondvarSignal()</i>	<i>pthread_cond_broadcast()</i> , <i>pthread_cond_signal()</i>	Wake up condvar-blocked threads.
<i>SyncMutexLock()</i>	<i>pthread_mutex_lock()</i> , <i>pthread_mutex_trylock()</i>	Lock a mutex.
<i>SyncMutexUnlock()</i>	<i>pthread_mutex_unlock()</i>	Unlock a mutex.
<i>SyncSemPost()</i>	<i>sem_post()</i>	Post a semaphore.
<i>SyncSemWait()</i>	<i>sem_wait()</i> , <i>sem_trywait()</i>	Wait on a semaphore.

QNX Neutrino IPC

IPC plays a fundamental role in the transformation of QNX Neutrino 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 QNX Neutrino, 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

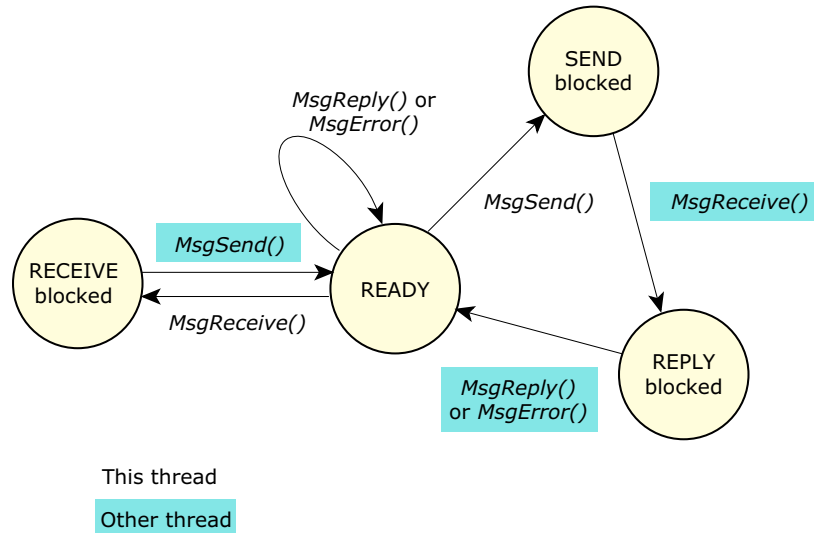
These services can be selected by the designer on the basis of bandwidth requirements, the need for queuing, network transparency, etc. The tradeoff can be complex, but the flexibility is useful.

As part of the engineering effort that went into defining the QNX Neutrino 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

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()*.



State changes in a send-receive-reply transaction.

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()*.

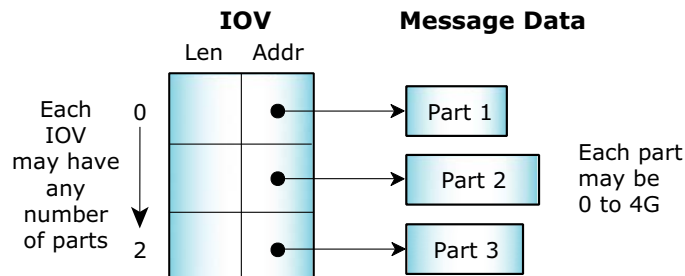
Message copying

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.

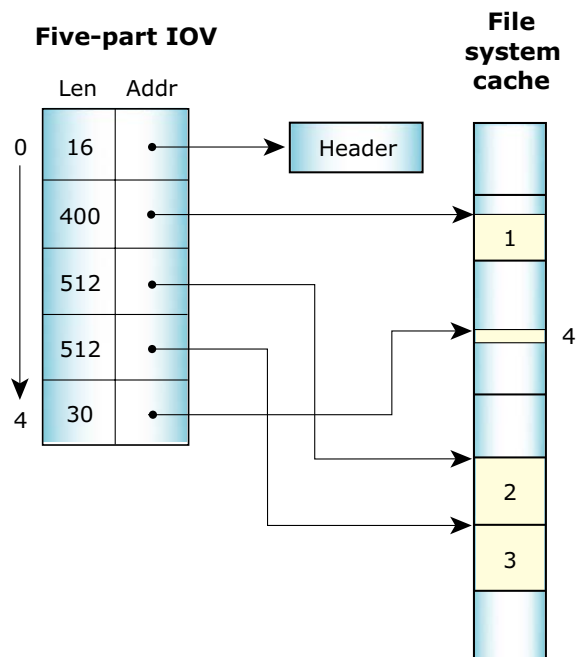
Multipart transfers allow messages that have a header block separate from the data block to be sent without performance-consuming copying of the data to create a contiguous message. In addition, if the underlying data structure is a ring buffer, specifying a three-part message will allow a header and 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.



A multipart transfer.

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 5-part message:



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;
    }

    off64_t tell64(int fd) {
        return lseek64(fd, 0, SEEK_CUR);
    }

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

    off_t tell(int fd) {
        return lseek64(fd, 0, SEEK_CUR);
    }

```

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
<i>MsgSend()</i>	simple	simple
<i>MsgSendsv()</i>	simple	IOV
<i>MsgSendsv()</i>	IOV	simple
<i>MsgSendv()</i>	IOV	IOV

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

IOV	Simple direct
<i>MsgReceivev()</i>	<i>MsgReceive()</i>
<i>MsgReceivePulsev()</i>	<i>MsgReceivePulse()</i>
<i>MsgReplyv()</i>	<i>MsgReply()</i>
<i>MsgReadv()</i>	<i>MsgRead()</i>
<i>MsgWritev()</i>	<i>MsgWrite()</i>

Channels and connections

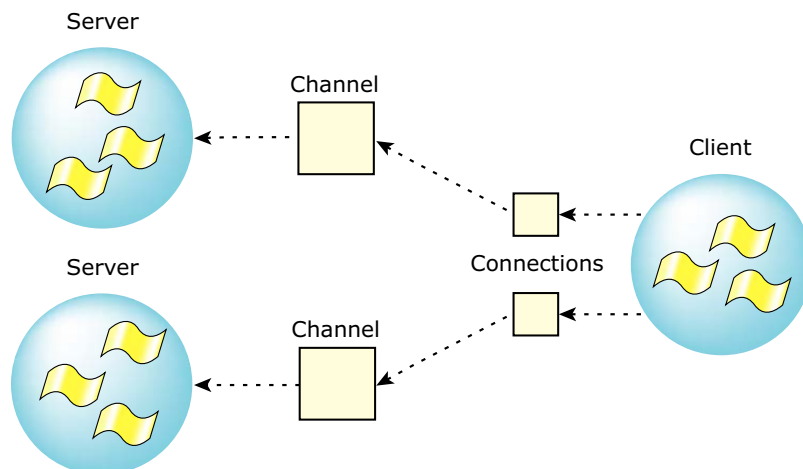
In QNX Neutrino, message passing is directed towards channels and connections, rather than targeted directly from thread to thread. A thread that wishes to receive messages first creates a channel; another

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
<i>ChannelCreate()</i>	Create a channel to receive messages on.
<i>ChannelDestroy()</i>	Destroy a channel.
<i>ConnectAttach()</i>	Create a connection to send messages on.
<i>ConnectDetach()</i>	Detach a connection.



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(&iiov, &msg, sizeof(msg));
for(;;) {
    rcv_id = MsgReceivev( chid, &iiov, parts, &info );

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

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

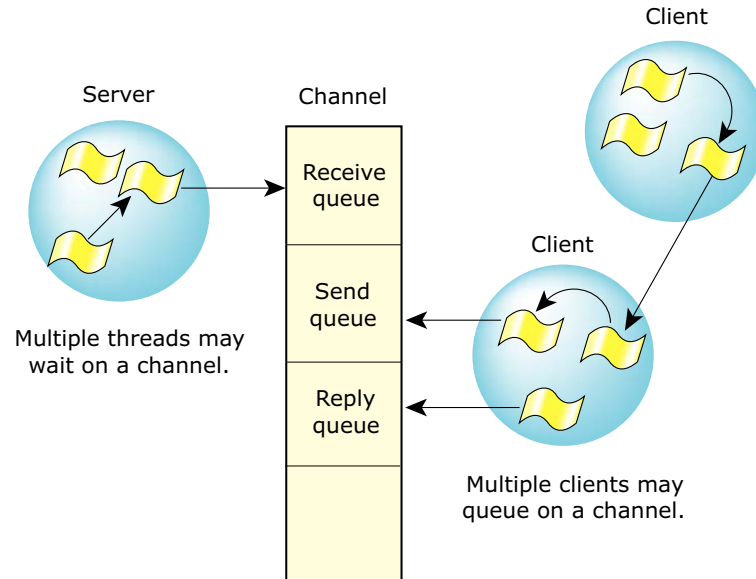
This loop allows the thread to receive messages from any thread that had a connection to the channel.

The channel has three queues associated with it:

- one queue for threads waiting for messages
- one queue for threads that have sent a message that hasn't yet been received

- one queue for threads that have sent a message that has been received, but not yet replied to.

While in any of these queues, the waiting thread is blocked (i.e. RECEIVE-, SEND-, or REPLY-blocked).

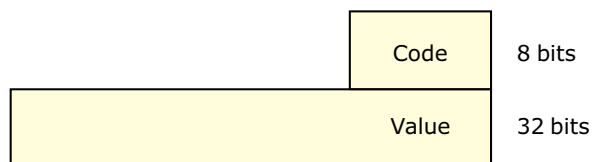


Threads blocked while in a channel queue.

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.



Pulses pack a small payload.

Priority inheritance

A server process receives messages in priority order. As the threads within the server receive requests, they then inherit the priority of the sending thread (but not the scheduling algorithm). 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.

Message-passing API

The message-passing API consists of the following functions:

Function	Description
<i>MsgSend()</i>	Send a message and block until reply.
<i>MsgReceive()</i>	Wait for a message.
<i>MsgReceivePulse()</i>	Wait for a tiny, nonblocking message (pulse).
<i>MsgReply()</i>	Reply to a message.
<i>MsgError()</i>	Reply only with an error status. No message bytes are transferred.
<i>MsgRead()</i>	Read additional data from a received message.
<i>MsgWrite()</i>	Write additional data to a reply message.

continued...

Function	Description
<i>MsgInfo()</i>	Obtain info on a received message.
<i>MsgSendPulse()</i>	Send a tiny, nonblocking message (pulse).
<i>MsgDeliverEvent()</i>	Deliver an event to a client.
<i>MsgKeyData()</i>	Key a message to allow security checks.

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.

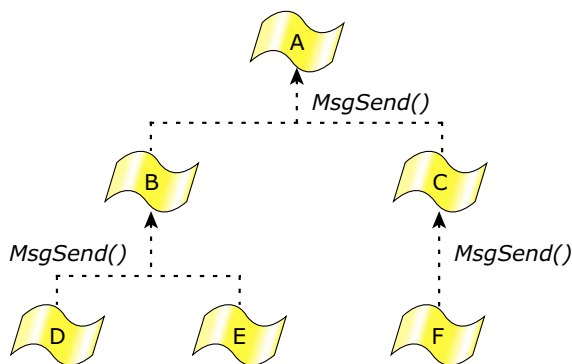
With a synchronous, nonqueued system architecture built around Send/Receive/Reply, robust application architectures can be very readily implemented and delivered.

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:



Threads should always send up to higher-level threads.

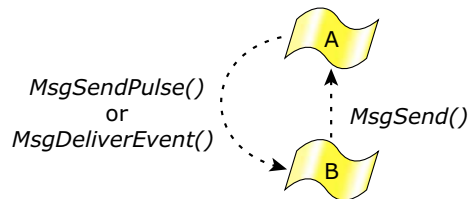
Here the threads at any given level in the hierarchy never send to each other, but send only upwards instead.

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.)

QNX Neutrino 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.



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 advance 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-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.

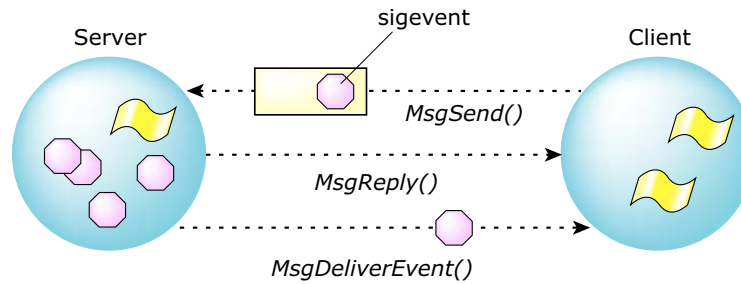
The events encountered by an executing thread can come from any of three sources:

- 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,” to the server to hang on to until later. When the server needs to notify the client thread, it will invoke *MsgDeliverEvent()* and the microkernel will set the event type encoded within the cookie upon the client thread.



The client sends a **sigevent** to the server.

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 it. 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_OUT_OF_BAND` — 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. QNX Neutrino 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
<i>SignalKill()</i>	<i>kill()</i> , <i>pthread_kill()</i> , <i>raise()</i> , <i>sigqueue()</i>	Set a signal on a process group, process, or thread.
<i>SignalAction()</i>	<i>sigaction()</i>	Define action to take on receipt of a signal.
<i>SignalProcmask()</i>	<i>sigprocmask()</i> , <i>pthread_sigmask()</i>	Change signal blocked mask of a thread.
<i>SignalSuspend()</i>	<i>sigsuspend()</i> , <i>pause()</i>	Block until a signal invokes a signal handler.

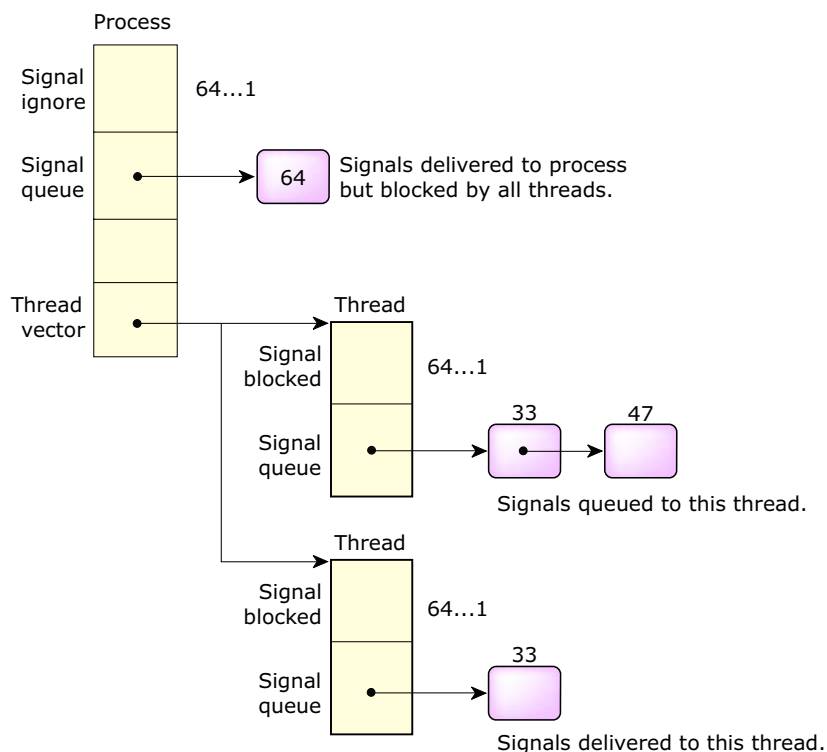
continued...

Microkernel call	POSIX call	Description
<i>SignalWaitinfo()</i>	<i>sigwaitinfo()</i>	Wait for signal and return info on it.

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

- 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 un-ignored signal targeted at a thread will be delivered to that thread alone.
- An un-ignored 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 multi-threaded 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.



Signal delivery.

The POSIX standard includes the concept of queued realtime signals. QNX Neutrino 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.

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.

This signal mechanism is used by Photon to wait for events and by the *select()* function to wait for I/O from multiple servers. Of the eight special signals, the first two have been given special names for this use.

```
#define SIGSELECT    (SIGRTMAX + 1)
```

```
#define SIGPHOTON (SIGRTMAX + 2)
```

Summary of signals

Signal	Description
SIGABRT	Abnormal termination signal such as issued by the <i>abort()</i> function.
SIGALRM	Timeout signal such as issued by the <i>alarm()</i> function.
SIGBUS	Indicates a memory parity error (QNX-specific interpretation). Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGCHLD	Child process terminated. The default action is to ignore the signal.
SIGCONT	Continue if HELD. The default action is to ignore the signal if the process isn't HELD.
SIGDEADLK	Mutex deadlock occurred. If you haven't called <i>SyncMutexEvent()</i> , and if the conditions that would cause the kernel to deliver the event occur, then the kernel delivers a SIGDEADLK instead.
SIGEMT	EMT instruction (emulator trap).
SIGFPE	Erroneous arithmetic operation (integer or floating point), such as division by zero or an operation resulting in overflow. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGHUP	Death of session leader, or hangup detected on controlling terminal.

continued...

Signal	Description
SIGILL	Detection of an invalid hardware instruction. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGINT	Interactive attention signal (Break).
SIGIOT	IOT instruction (not generated on x86 hardware).
SIGKILL	Termination signal — should be used only for emergency situations. <i>This signal cannot be caught or ignored.</i>
SIGPIPE	Attempt to write on a pipe with no readers.
SIGPOLL	Pollable event occurred.
SIGQUIT	Interactive termination signal.
SIGSEGV	Detection of an invalid memory reference. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGSTOP	Stop process (the default). <i>This signal cannot be caught or ignored.</i>
SIGSYS	Bad argument to system call.
SIGTERM	Termination signal.
SIGTRAP	Unsupported software interrupt.
SIGTSTP	Stop signal generated from keyboard.
SIGTTIN	Background read attempted from control terminal.
SIGTTOU	Background write attempted to control terminal.
SIGURG	Urgent condition present on socket.
SIGUSR1	Reserved as application-defined signal 1.

continued...

Signal	Description
SIGUSR2	Reserved as application-defined signal 2.
SIGWINCH	Window size changed.

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.



To use POSIX message queues in QNX Neutrino, the message queue server must be running. QNX Neutrino has two implementations of message queues:

- a “traditional” implementation that uses the **mqqueue** resource manager (see the Resource Managers chapter in this book)
- an alternate implementation that uses the **mq** server and asynchronous messages.

For more information about these implementations, see the *Utilities Reference*.

Unlike our inherent message-passing primitives, the POSIX message queues reside *outside* the kernel.

Why use POSIX message queues?

POSIX message queues provide a familiar interface for many realtime programmers. They are similar to the “mailboxes” found in many realtime executives.

There’s a fundamental difference between our messages and POSIX message queues. Our messages block — they copy their data directly between the address spaces of the processes sending the messages. POSIX messages 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 would likely use message queues in a design where a number of named queues will be operated on by a variety of processes over time.

For raw performance, POSIX message queues will be *slower* than QNX Neutrino native messages for transferring data. However, the flexibility of queues may make this small performance penalty worth the cost.

File-like interface

Message queues resemble files, at least as far as their interface is concerned. You open a message queue with `mq_open()`, close it with `mq_close()`, and destroy it with `mq_unlink()`. And to put data into (“write”) and take it out of (“read”) a message queue, you use `mq_send()` and `mq_receive()`.

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 will 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:

<i>mq_open()</i> name:	Pathname of message queue:
<i>/data</i>	<i>/dev/mqueue/data</i>
<i>/acme/data</i>	<i>/dev/mqueue/acme/data</i>
<i>/qnx/data</i>	<i>/dev/mqueue/qnx/data</i>

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.

Message-queue functions

POSIX message queues are managed via the following functions:

Function	Description
<i>mq_open()</i>	Open a message queue.
<i>mq_close()</i>	Close a message queue.
<i>mq_unlink()</i>	Remove a message queue.
<i>mq_send()</i>	Add a message to the message queue.
<i>mq_receive()</i>	Receive a message from the message queue.
<i>mq_notify()</i>	Tell the calling process that a message is available on a message queue.
<i>mq_setattr()</i>	Set message queue attributes.
<i>mq_getattr()</i>	Get message queue attributes.

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 CD-ROM 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
<i>shm_open()</i>	Open (or create) a shared-memory region.
<i>close()</i>	Close a shared-memory region.
<i>mmap()</i>	Map a shared-memory region into a process's address space.
<i>munmap()</i>	Unmap a shared-memory region from a process's address space.
<i>mprotect()</i>	Change protections on a shared-memory region.
<i>msync()</i>	Synchronize memory with physical storage.
<i>shm_ctl()</i>	Give special attributes to a shared-memory object.
<i>shm_unlink()</i>	Remove a shared-memory region.

POSIX shared memory is implemented in QNX Neutrino 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.

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

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.

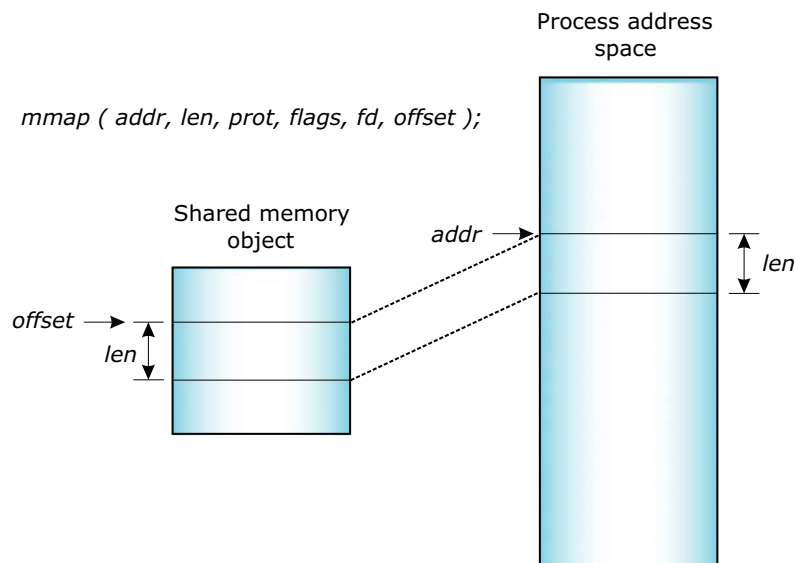
The *mmap()* function is defined as follows:

```
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.



Arguments to 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 reign 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.

continued...

Manifest	Description
PROT_READ	Memory may be read.
PROT_WRITE	Memory may be written.

The PROT_NOCACHE manifest should be used when a shared-memory region is used 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 is shared by the calling processes.
MAP_PRIVATE	The mapping is private to the calling process. It allocates system RAM and makes a copy of the object.
MAP_ANON	Similar to MAP_PRIVATE, but the <i>fd</i> parameter isn’t used (should be set to NOFD), and the allocated memory is zero-filled.

The MAP_SHARED type is the one to use for setting up shared memory between processes. The other types have more specialized uses. For example, MAP_ANON can be used as the basis for a page-level memory allocator.

A number of flags may be ORed into the above type to further define the mapping. These are described in detail in the *mmap()* entry in the *Library Reference*. A few of the more interesting flags are:

Map type modifier	Description
MAP_FIXED	Map object to the address specified by <i>where_i_want_it</i> . 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 <i>fd</i> parameter should be set to NOFD. When used with MAP_SHARED, the <i>offset_within_shared_memory</i> 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). MAP_NOX64K and MAP_BELOW16M are used to further define the MAP_ANON allocated memory and address limitations present in some forms of DMA.
MAP_NOX64K	(x86-specific) Used with MAP_PHYS MAP_ANON. The allocated memory area will not cross a 64K boundary. This is required for the old 16-bit PC DMA.
MAP_BELOW16M	(x86-specific) Used with MAP_PHYS MAP_ANON. The allocated memory area will reside in physical memory below 16M. This is necessary when using DMA with ISA bus devices.

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 share memory with hardware such as video memory:

```
/* Map in VGA display memory */
addr = mmap(0, 65536, PROT_READ|PROT_WRITE,
            MAP_PHYS|MAP_SHARED, NOFD, 0xa0000);
```

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_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.

Pipes and FIFOs



To use pipes or FIFOs in QNX Neutrino, 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	<i>pipe()</i> or <i>popen()</i> 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	<i>mkfifo()</i> function
Remove FIFOs from within the shell	rm utility
Remove FIFOs from within programs	<i>remove()</i> or <i>unlink()</i> function

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 January 2554, but note that POSIX-compliant code may be limited to the year 2038. The internal date and time representation reaches its maximum value in 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. On a Pentium processor, this counter increments on each clock cycle. A 100 MHz Pentium would have a cycle time of 1/100,000,000 seconds (10 nanoseconds). Other CPU architectures have similar instructions.

On processors that don't implement such an instruction in hardware (e.g. a 386), 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 `SYPAGE_ENTRY(qtime)->cycles_per_sec` field gives the number of *ClockCycles()* increments in one second.

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.

Microkernel call	POSIX call	Description
<i>ClockTime()</i>	<i>clock_gettime()</i> , <i>clock_settime()</i>	Get or set the time of day (using a 64-bit value in nanoseconds ranging from 1970 to 2554).
<i>ClockAdjust()</i>	N/A	Apply small time adjustments to synchronize clocks.
<i>ClockCycles()</i>	N/A	Read a 64-bit free-running high-precision counter.
<i>ClockPeriod()</i>	<i>clock_getres()</i>	Get or set the period of the clock.
<i>ClockId()</i>	<i>clock_getcpuclockid()</i> , <i>pthread_getcpuclockid()</i>	Return an integer that's passed to <i>ClockTime()</i> as a clockid_t .

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

QNX Neutrino 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 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
<i>TimerAlarm()</i>	<i>alarm()</i>	Set a process alarm.
<i>TimerCreate()</i>	<i>timer_create()</i>	Create an interval timer.
<i>TimerDestroy()</i>	<i>timer_delete()</i>	Destroy an interval timer.
<i>TimerGettime()</i>	<i>timer_gettime()</i>	Get time remaining on an interval timer.
<i>TimerGetoverrun()</i>	<i>timer_getoverrun()</i>	Get number of overruns on an interval timer.
<i>TimerSettime()</i>	<i>timer_settime()</i>	Start an interval timer.
<i>TimerTimeout()</i>	<i>sleep()</i> , <i>nanosleep()</i> , <i>sigtimedwait()</i> , <i>pthread_cond_timedwait()</i> , <i>pthread_mutex_trylock()</i> , <i>intr_timed_wait()</i>	Arm a kernel timeout for any blocking state.

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.

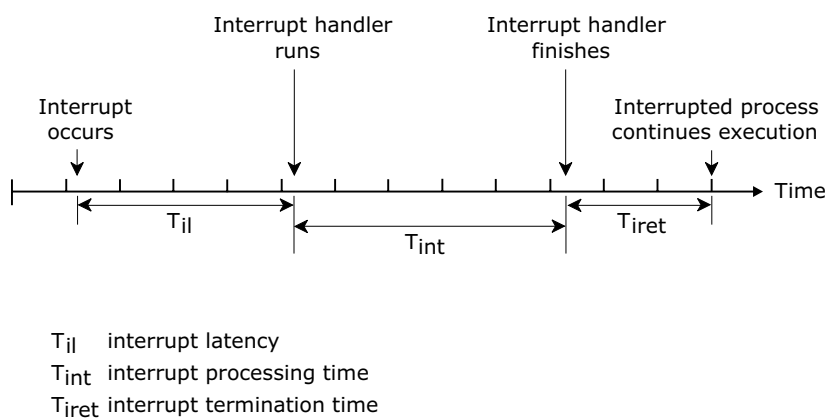


Latency times can vary significantly, depending on the speed of the processor and other factors. For more information, visit our website (www.qnx.com).

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.



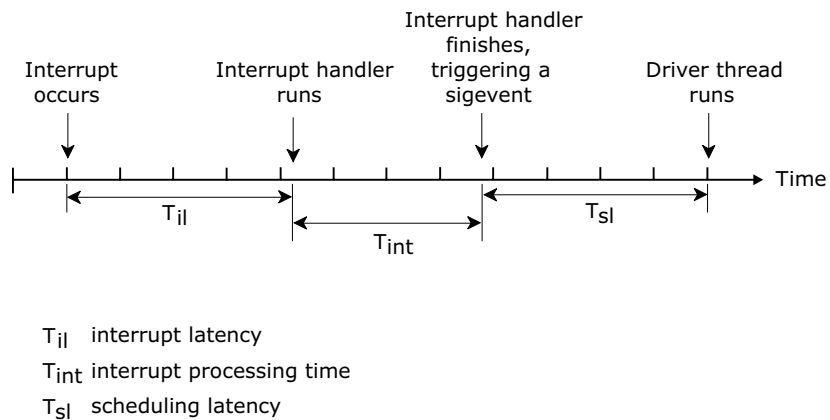
Interrupt handler simply terminates.

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.



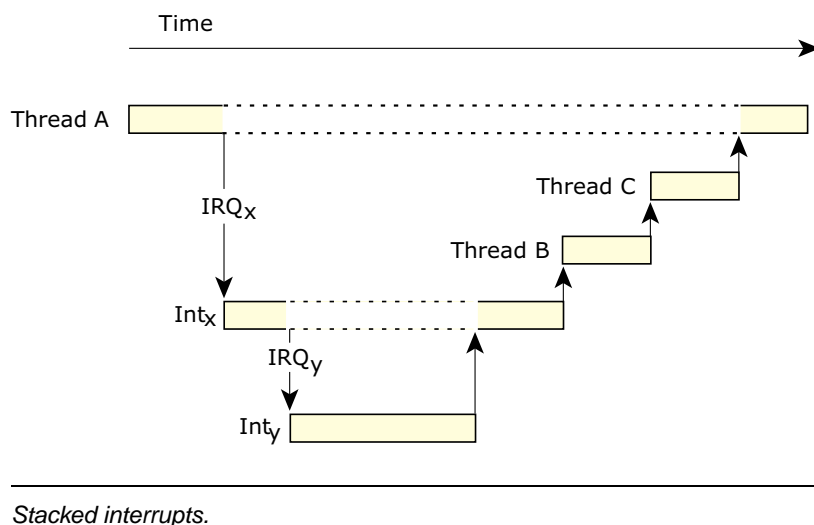
Interrupt handler terminates, returning an event.

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

QNX Neutrino 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.



Interrupt calls

The interrupt-handling API includes the following kernel calls:

Function	Description
<i>InterruptAttach()</i>	Attach a local function to an interrupt vector.
<i>InterruptAttachEvent()</i>	Generate an event on an interrupt, which will ready a thread. No user interrupt handler runs. This is the preferred call.
<i>InterruptDetach()</i>	Detach from an interrupt using the ID returned by <i>InterruptAttach()</i> or <i>InterruptAttachEvent()</i> .
<i>InterruptWait()</i>	Wait for an interrupt.
<i>InterruptEnable()</i>	Enable hardware interrupts.
<i>InterruptDisable()</i>	Disable hardware interrupts.
<i>InterruptMask()</i>	Mask a hardware interrupt.
<i>InterruptUnmask()</i>	Unmask a hardware interrupt.
<i>InterruptLock()</i>	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 <i>InterruptDisable()</i> and should be used in its place.
<i>InterruptUnlock()</i>	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.



For more information on *InterruptLock()* and *InterruptUnlock()*, see “Critical sections” in the chapter on SMP in this guide.

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;

    // Request I/O privileges
    ThreadCtl( _NTO_TCTL_IO, 0 );

    // Initialize event structure
    event.sigev_notify = SIGEV_INTR;

    // 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
<i>InterruptHookIdle()</i>	When the kernel has no active thread to schedule, it will run the idle thread, which can upcall to a user handler. This handler can perform hardware-specific power-management operations.
<i>InterruptHookTrace()</i>	This function attaches a pseudo interrupt handler that can receive trace events from the instrumented kernel.



Chapter 3

The Instrumented Microkernel

In this chapter...

Introduction	107
Instrumentation at a glance	107
Event control	108
Data interpretation	110
Proactive tracing	112



Introduction

An instrumented version of the QNX Neutrino 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 30K 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.

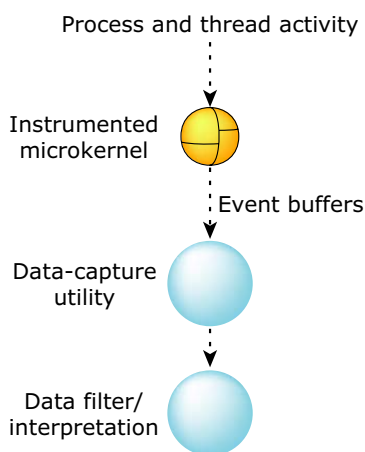
The instrumented module is nonintrusive — you don't have to modify a program's source code in order to monitor how that program interacts with the kernel. You can trace as many or as few interactions (e.g. kernel calls, state changes, and other system activities) as you want between the kernel and any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

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.



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
# of parts to send	4 bytes
# of parts to receive	4 bytes

continued...

Wide mode data	Number of bytes for the event
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, PowerPC, 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
.
.
.

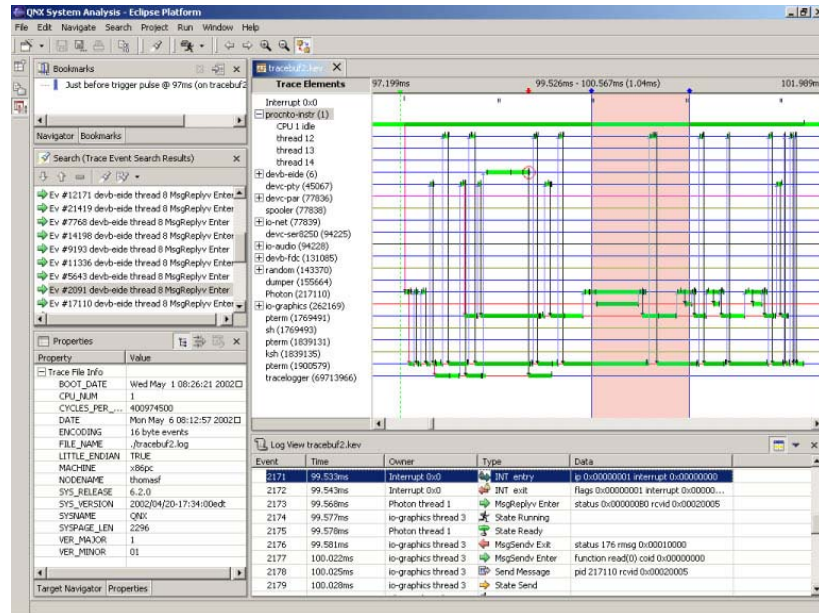
```

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.



The IDE helps you visualize system activity.

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_INSERTUSEREVENT, 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 4

SMP

In this chapter...

Introduction	117
Booting an x86 SMP system	118
Booting a PowerPC or MIPS SMP system	119
How the SMP microkernel works	119
Critical sections	121



Introduction

SMP (Symmetrical Multi-Processing) is typically associated with high-end operating systems such as UNIX and NT running on high-end servers. These large monolithic systems tend to be quite complex, the result of many man-years of development. Since these large kernels contain the bulk of all OS services, the changes to support SMP are extensive, usually requiring large numbers of modifications and the use of specialized spinlocks throughout the code.

QNX Neutrino, on the other hand, contains a very small microkernel surrounded by processes that act as resource managers, providing services such as filesystems, character I/O, and networking. By modifying the microkernel alone, all other OS services will gain full advantage of SMP *without the need for coding changes*. If these service-providing processes are multi-threaded, their many threads will be scheduled among the available processors. Even a single-threaded server would also benefit from an SMP system, because its thread would be scheduled on the available processors beside other servers and client processes.

SMP versions of `procnto*`

As a testament to this microkernel approach, the SMP-enabled QNX Neutrino kernel/process manager adds only a few kilobytes of additional code. The SMP versions are designed for these main processor families:

- PowerPC (e.g. `procnto-600-smp`).
- MIPS (`procnto-smp`)
- x86 (`procnto-smp`)

The x86 version can boot on any system that conforms to the Intel MultiProcessor Specification (MP Spec) with up to eight Pentium (or better) processors. QNX Neutrino also supports Intel's new Hyper-Threading Technology found in P4 and Xeon processors.

The **procnto-smp** manager will also function on a single non-SMP system. With the cost of building a dual-processor Pentium motherboard very nearly the same as a single-processor motherboard, it's possible to deliver cost-effective solutions that can be scaled in the field by the simple addition of a second CPU. The fact that the OS itself is only a few kilobytes larger also allows SMP to be seriously considered for small CPU-intensive embedded systems, not just high-end servers.

The PowerPC and MIPS versions of the SMP-enabled kernel deliver full SMP support (e.g. cache-coherency, inter-processor interrupts, etc.) on appropriate PPC and MIPS hardware. The PPC version supports any SMP system with 7xx or 74xx series processors, as in such reference design platforms as the Motorola MVP or the Marvell EV-64260-2XMPC7450 SMP Development System. The MIPS version supports such systems as the dual-core Broadcom BCM1250 processor.

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-bios** program is designed to work on systems compatible with the Intel MP Spec (version 1.4 or later). 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 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-bios** 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.

Booting a PowerPC or MIPS SMP system

On a PPC or MIPS SMP system, the boot sequence is similar to that of an x86, but a specific startup program (e.g. **startup-mvp**, **startup-bcm1250**) will be used instead. Specifically, the PPC-specific startup is responsible for:

- determining the number of processors
- initializing each additional processor
- initializing the IRQ, IPI, system controller, etc.

For each additional processor found, the startup code will:

- initialize the processor
- initialize the MMU
- initialize the caches
- 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 algorithm follows the same rules as on a uniprocessor system. That is, the highest-priority thread will be running on the 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 *low-priority* threads, with an eye towards optimizing cache usage and minimizing thread migration. In any case, the realtime scheduling rules that were in place on a uniprocessor system are guaranteed to be upheld on an SMP system.

Hard processor affinity

QNX Neutrino also supports the concept of hard processor affinity through the kernel call `ThreadCtl(_NTO_TCTL_RUNMASK, runmask)`. Each set bit in *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 careful use of this primitive, 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.

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 preemptable 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 preemptable 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.

Inter-processor interrupts (IPIs)

The processors communicate with each other through IPIs (inter-processor 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 QNX Neutrino, 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 (by setting the processor affinity to 1 via the *ThreadCtl()* function).

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 *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(int_{spin_t}* *spinlock*)

Release a lock and reenale interrupts.

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



Chapter 5

Process Manager

In this chapter...

Introduction	127
Process management	127
Memory management	133
Pathname management	140



Introduction

In QNX Neutrino, the microkernel is paired with the Process Manager in a single module (**procnto**). This module is required for all runtime systems.

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

Process primitives

There are four process primitives:

<i>spawn()</i>	POSIX
<i>fork()</i>	POSIX
<i>vfork()</i>	UNIX BSD extension
<i>exec*()</i>	POSIX

spawn()

The *spawn()* call was introduced by POSIX. The function creates a child process by directly specifying an executable to load. To those familiar with UNIX systems, the 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.

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
```

The POSIX *spawn()* function gives control over four classes of environment inheritance, which are often adjusted when creating a new child process:

- file descriptors

- process group ID
- signal mask
- ignored signals.

Our implementation also allows you to change:

- node to create process on
- scheduling policy
- scheduling parameters (priority)
- maximum stack size.

There are two forms of the POSIX *spawn()* function:

<i>spawn()</i>	Spawn with explicit specified path.
<i>spawnp()</i>	Search the current PATH and invoke <i>spawn()</i> with the first matching executable.

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

<i>spawnl()</i>	Spawn with command line provided as inline arguments.
<i>spawnle()</i>	<i>spawnl()</i> with explicitly passed environment variables.
<i>spawnlp()</i>	<i>spawnp()</i> that follows the command search path.
<i>spawnlpe()</i>	<i>spawnlp()</i> with explicitly passed environment variables.
<i>spawnv()</i>	Spawn with command line pointed to by an array of pointers.
<i>spawnve()</i>	<i>spawnv()</i> with explicitly passed environment variables.

spawnvp() *spawnv()* that follows the command search path.
spawnvpe() *spawnvp()* with explicitly passed environment variables.

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

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 *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 lists some resources that 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, which combines 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 *spawn()* API.



Note that *fork()* should be called from a process containing only a single thread.

vfork()

The *vfork()* function (which should also be called only from a single-threaded process) is useful when the purpose of *fork()* would have been to create a new system context for a call to one of the *exec*()* functions. The *vfork()* function differs from *fork()* in that the child doesn't get a copy of the calling process's data. Instead, it borrows the calling process's memory and thread of control until a call to one of the *exec*()* functions is made. The calling process is suspended while the child is using its resources.

The *vfork()* child can't return from the procedure that called *vfork()*, since the eventual return from the parent *vfork()* would then return to a stack frame that no longer existed.

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:

<i>execl()</i>	Exec with command line provided as inline arguments.
<i>execle()</i>	<i>execl()</i> with explicitly passed environment variables.
<i>execlp()</i>	<i>execl()</i> that follows the command search path.
<i>execlpe()</i>	<i>execlp()</i> with explicitly passed environment variables.
<i>execv()</i>	<i>execl()</i> with command line pointed to by an array of pointers.
<i>execve()</i>	<i>execv()</i> with explicitly passed environment variables.
<i>execvp()</i>	<i>execv()</i> that follows the command search path.
<i>execvpe()</i>	<i>execvp()</i> with explicitly passed environment variables.

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

Process loading

Processes loaded from a filesystem using either the *exec*()* or *spawn()* calls are in ELF format. If the filesystem is on a block-oriented device, the code and data are loaded into main memory.

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 on 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 overwrite 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.”

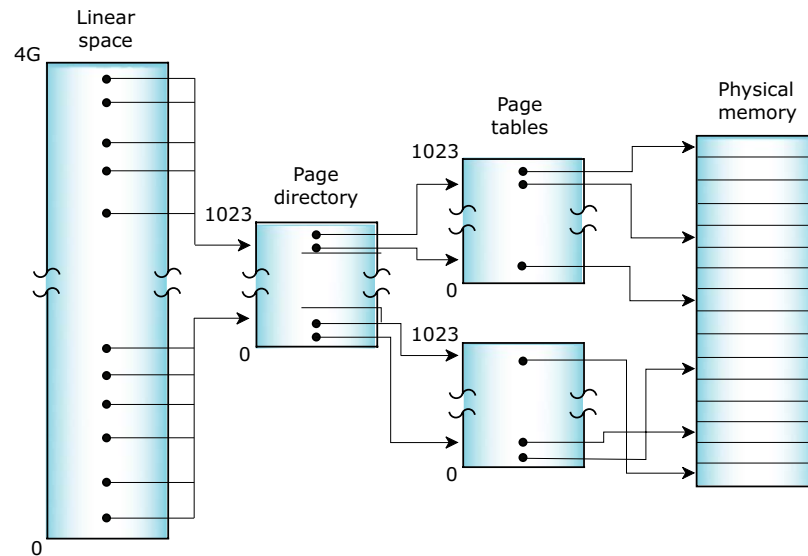
With an MMU enabled, the OS can abort the process the instant the memory-access violation occurs, providing immediate feedback to the programmer instead of mysteriously crashing the system some time later. The OS can then provide the location of the errant instruction in the failed process, or position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

Memory Management Units (MMUs)

A typical MMU operates by dividing physical memory into a number of 4K *pages*. The hardware within the processor then makes use of 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.



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 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 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 dumpfiles 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 dumpfiles 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. Making use of the memory protection delivered by integrated MMUs in the embedded systems we build is a good step in that direction.

Full-protection model

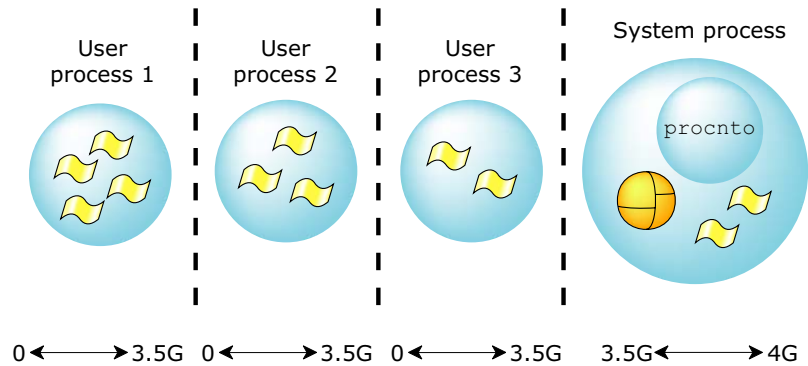
Our full-protection model relocates all code in the image into a new virtual space, enabling the MMU hardware and setting up the initial page-table mappings. This allows **procnto** to start in a correct, MMU-enabled environment. The process manager will then take over this environment, changing the 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, which spans to 2 or 3.5 Gigabytes (depending on the CPU). 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, SH-4, ARM, and MIPS processors, but not on the PowerPC, where the space from 0 to 1G is reserved for system processes.



Full protection VM (on an x86).

The memory cost per process may increase by 4K to 8K for each process's page tables. Note that this memory model supports the POSIX *fork()* call.

Pathname management

Domains of authority

I/O resources are *not* 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. 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/	The running processes, each represented by their process ID (PID).
/dev/zero	A device that always returns zero. Used for allocating zero-filled pages using the <i>mmap()</i> 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. If the regions are overlapping, the longest match wins. For example, suppose we have these prefixes registered:

continued...

```

/          QNX4 disk-based filesystem (fs-qnx4.so)

/          QNX4 disk-based filesystem (fs-qnx4.so)
/dev/ser1  serial device manager (devc-ser*)
/dev/ser2  serial device manager (devc-ser*)
/dev/hd0   raw disk volume (devb-eide.so)

```

The filesystem manager has registered a prefix for a mounted QNX 4 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:
<code>/dev/ser1</code>	<code>/dev/ser1</code>	<code>devc-ser*</code>
<code>/dev/ser2</code>	<code>/dev/ser2</code>	<code>devc-ser*</code>
<code>/dev/ser</code>	<code>/</code>	<code>fs-qnx4.so</code>
<code>/dev/hd0</code>	<code>/dev/hd0</code>	<code>devb-eide.so</code>
<code>/usr/jhsmith/test</code>	<code>/</code>	<code>fs-qnx4.so</code>

Consider another example involving three servers:

Server A	A QNX 4 filesystem. Its mountpoint is <code>/</code> . It contains the files <code>bin/true</code> and <code>bin/false</code> .
Server B	A flash filesystem. Its mountpoint is <code>/bin</code> . It contains the files <code>ls</code> and <code>echo</code> .
Server C	A single device that generates numbers. Its mountpoint is <code>/dev/random</code> .

At this point, the process manager's internal mount table would look like this:

Mountpoint	Server
/	Server A (QNX 4 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.

Single-device mountpoints

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 QNX 4 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 unioning 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 C 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.

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 <i>existing_file symbolic_Link</i>	Create a filesystem symbolic link.
ln -Ps <i>existing_file symbolic_Link</i>	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:

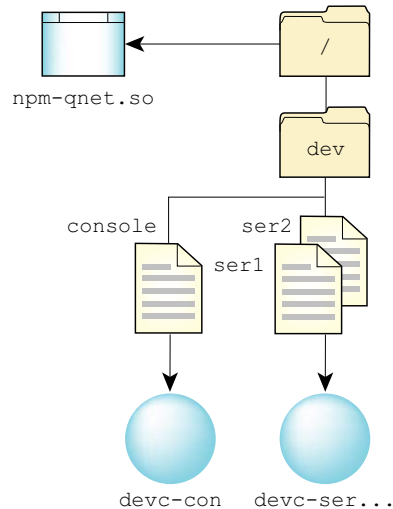
```
ln -Ps /net/neutron/bin /bin
```

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 **npm-qnet**. The **npm-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.

A note about `cd`

In some traditional UNIX systems, the `cd` (change directory) command modifies the pathname given to it if that pathname contains symbolic links. As a result, the pathname of the new current working directory — which you can display with `pwd` — may differ from the one given to `cd`.

In QNX Neutrino, however, `cd` doesn't modify the pathname — aside from collapsing `..` references. For example:

```
cd /usr/home/dan/test/../../doc
```

would result in a current working directory of `/usr/home/dan/doc`, even if some of the elements in the pathname were symbolic links.

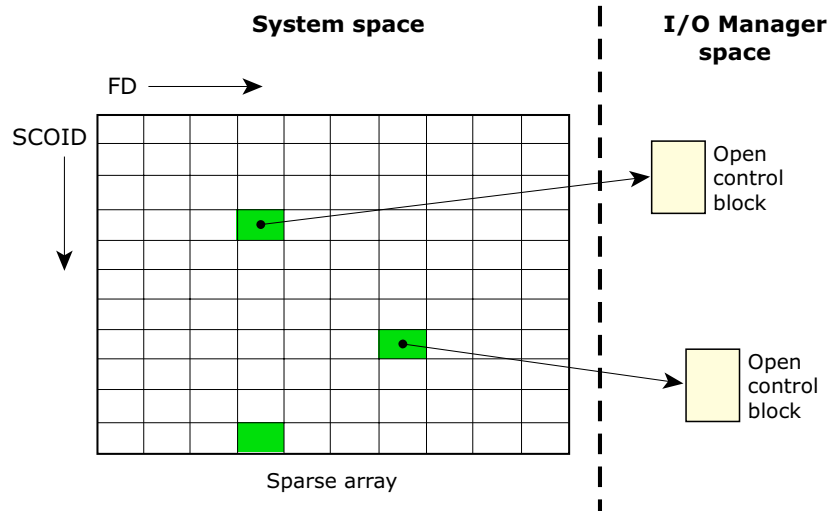
For more information about symbolic links and `..` references, see “QNX4 filesystem” in the Working with Filesystems chapter of the QNX Neutrino *User's Guide*.

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.

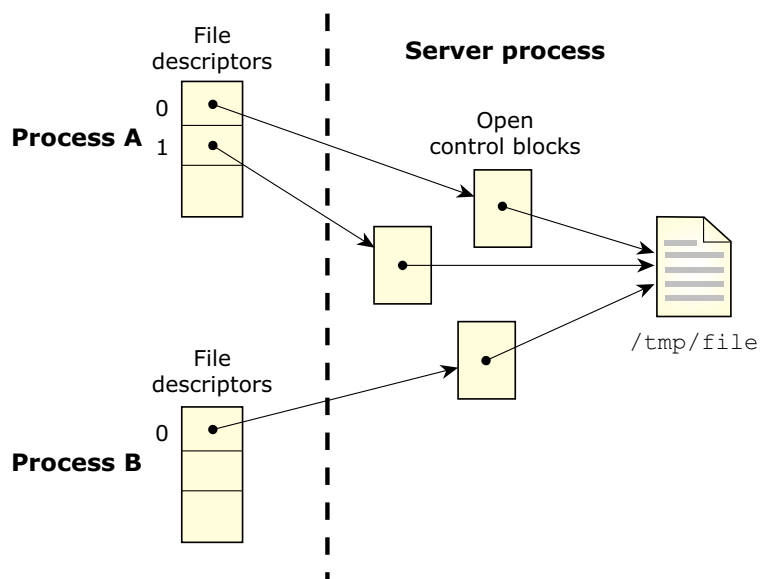


The SCOID and FD map to an OCB of an I/O Manager.

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.



Two processes open the same file.



FDs are a *process* resource, not a thread resource.

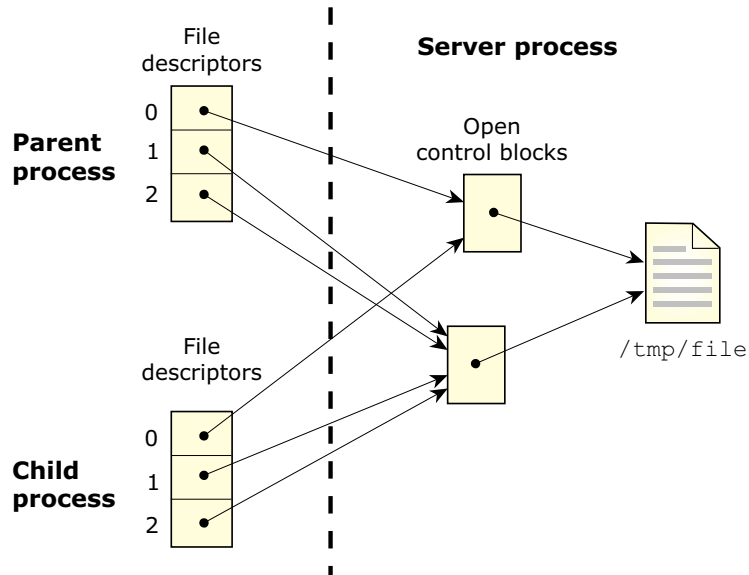
Several file descriptors in one or more processes can refer to the same OCB. This is accomplished by two means:

- 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 `vfork()`, `fork()`, 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.



A process opens a file twice.

You can prevent a file descriptor from being inherited when you *spawn()* or *exec*()* by calling the *fcntl()* function and setting the `FD_CLOEXEC` flag.



Chapter 6

Dynamic Linking

In this chapter...

Shared objects 155
How shared objects are used 157



Shared objects

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.

Statically linked

The term *statically linked* means that the program and the particular library that it's linked against are combined together by the linker at linktime. This means that the binding between the program and the particular library is fixed and known at linktime — well in advance of the program 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 term *dynamically linked* means that the program and the particular library it references are *not* combined together by the linker at linktime. 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 linktime, 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 toolchain 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 QNX 4 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-qnx4.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

QNX Neutrino 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 linktime, 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.

Linking view:	Execution view:
ELF header	ELF header
Program header table (optional)	Program header table
Section 1	Segment 1
...	
Section <i>n</i>	Segment 2
...	
...	...
Section header table	Section header table (optional)

Object file format: linking view and execution view.

ELF without COFF

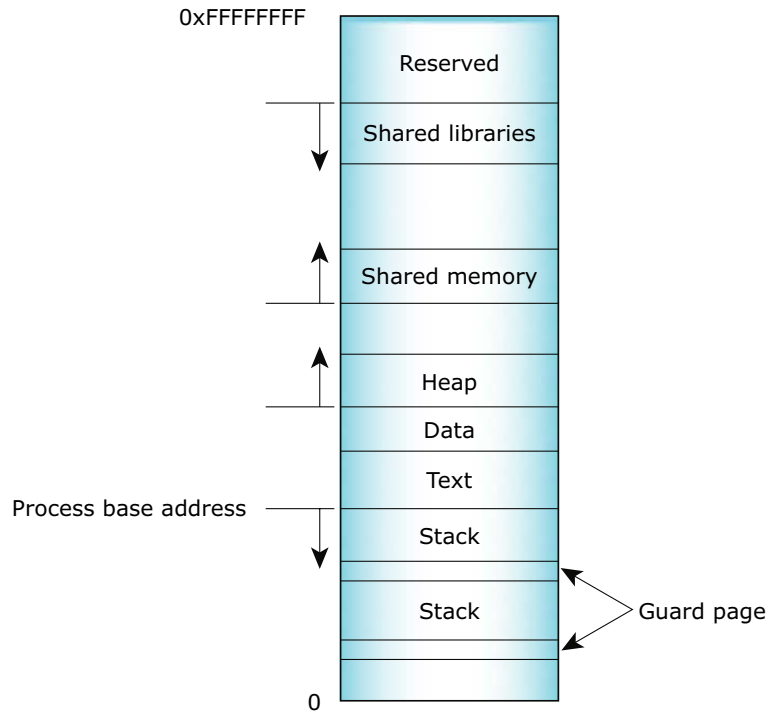
Most implementations of ELF loaders are derived from *COFF* (Common Object File Format) loaders; they use the linking view of the ELF objects at load time. This is inefficient because the program loader must load the executable using sections. A typical program could contain a large number of sections, 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.

The 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.



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 in the directories specified by **LD_LIBRARY_PATH** *only* if the program isn't marked as **setuid**.
- 2 If the shared library still isn't found, and if the executable's dynamic section contains a **DT_RPATH** tag, then the path specified by **DT_RPATH** is searched next.
- 3 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**. If none has been specified, then the default library path is set to the image filesystem's path.
- 4 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.

- 5 The shared library is then added to the internal list of all libraries that the process has loaded. The runtime linker maintains this list.
- 6 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 global list.
- 3 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 7

Resource Managers

In this chapter...

Introduction	167
What is a resource manager?	167
Resource manager architecture	173
Summary	180



Introduction

To give QNX Neutrino 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 QNX Neutrino 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 add-on 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, the utilities can be used for communicating with the resource managers.

For instance, the tiny TCP/IP protocol module contains resource-manager code that registers the name `/proc/ipstats`. If you open this name and read from it, the resource manager code responds with a body of text that describes the statistics for IP.

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 can type:

```
cat /proc/ipstats
```

The resource manager code in the TCP/IP protocol module responds with text such as:

```
Ttcpip Sep  5 2000 08:56:16

verbosity level 0
ip checksum errors: 0
udp checksum errors: 0
tcp checksum errors: 0

packets sent: 82
packets received: 82

lo0  : addr 127.0.0.1      netmask 255.0.0.0      up
DST: 127.0.0.0      NETMASK: 255.0.0.0      GATEWAY: lo0

TCP 127.0.0.1.1227      > 127.0.0.1.6000      ESTABLISHED snd    0 rcv    0
TCP 127.0.0.1.6000      > 127.0.0.1.1227      ESTABLISHED snd    0 rcv    0
TCP 0.0.0.0.6000        LISTEN
```

You could also use command-line utilities for a robot-arm driver. The driver could register the name, `/dev/robot/arm/angle`, and any writes to this device are interpreted 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,...`. 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 a flash driver toolkit is available that 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 QNX Neutrino 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 Writing a Resource Manager chapter in the *Programmer's 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.

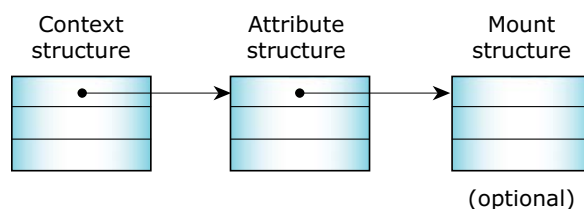
Second-level default message handling

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()*, *lseek()*, etc. *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



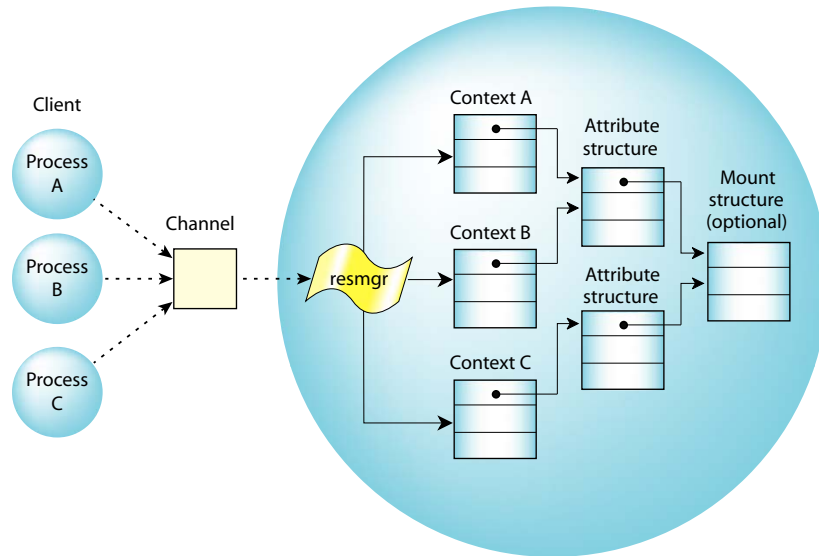
A resource manager is responsible for three data structures.

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:

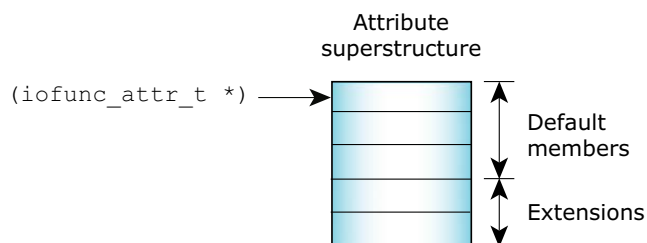


Multiple clients opening various devices.

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:



Encapsulation.

The library contains *iofunc_**(*)* default handlers for these client functions:

<i>chmod()</i>	<i>lock()</i>
<i>chown()</i>	<i>lseek()</i>
<i>close()</i>	<i>mmap()</i>
<i>devctl()</i>	<i>open()</i>
<i>fpathconf()</i>	<i>pathconf()</i>
<i>fseek()</i>	<i>stat()</i>
<i>fstat()</i>	<i>utime()</i>

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, QNX Neutrino offers the developer unprecedented flexibility and simplicity in developing “drivers” for new hardware — a critical feature for many embedded systems.

Chapter 8

Filesystems

In this chapter...

Introduction	183
Filesystem classes	184
Image filesystem	191
RAM “filesystem”	192
Embedded transaction filesystem (ETFS)	193
QNX4 filesystem	198
DOS Filesystem	199
CD-ROM filesystem	202
FFS3 filesystem	202
NFS filesystem	208
CIFS filesystem	209
Linux Ext2 filesystem	209
Virtual filesystems	209



Introduction

QNX Neutrino 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.

Most of 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.
- A filesystem running on one node is transparently accessible from any other node.

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 CD-ROM drives. This includes the QNX4, DOS, and CD-ROM 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.

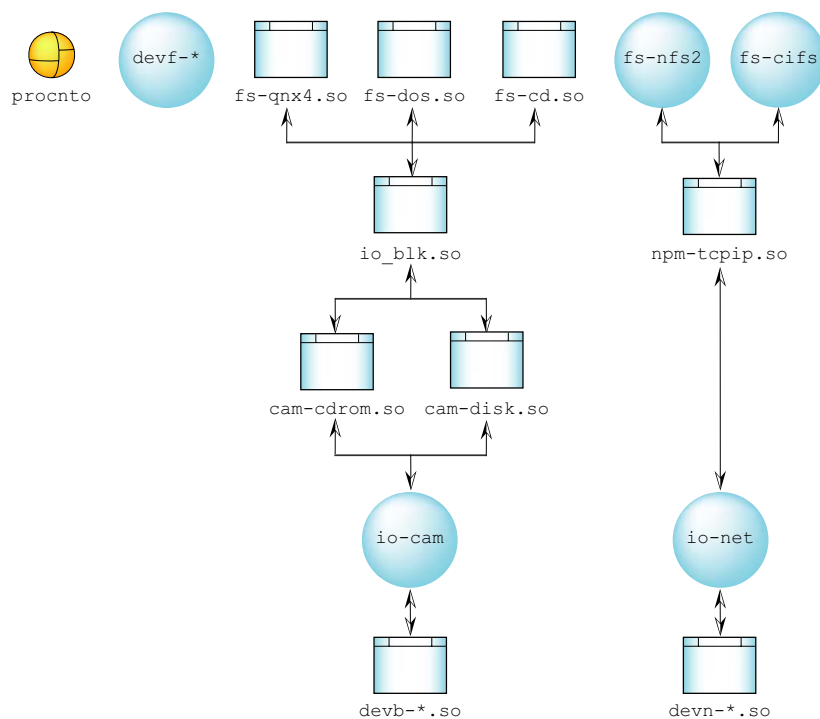
- Virtual QNX Neutrino provides the following virtual filesystems:
- Package Filesystem, which presents custom “views” of selected files and directories to a client
 - Inflator, a resource manager that sits in front of other filesystems and uncompresses files that were previously compressed (using the **deflate** utility).

Filesystems as shared libraries

Since it's common to run many filesystems under QNX Neutrino, 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:



QNX Neutrino filesystem layering.

As shown in this diagram, the filesystems, disk drivers, and `io-blk` are implemented as shared libraries (essentially passive blocks of code resident in memory), while `io-cam` is the active executable that calls into the libraries. In operation, the `io-cam` process starts first and invokes the block-level shared library (`io-blk.so`) as well as the disk driver shared libraries (`devb-*`). 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 (PCCard flash card, floppy disk, 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 (**io-blk**). This 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

QNX Neutrino 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 QNX partition and a DOS partition, while the second disk had only a QNX partition, then the default files would be:

<code>/dev/hd0</code>	First hard disk
<code>/dev/hd0t6</code>	DOS partition on first hard disk
<code>/dev/hd0t79</code>	QNX partition on first hard disk
<code>/dev/hd1</code>	Second hard disk
<code>/dev/hd1t79</code>	QNX 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)
6	DOS 4.0 (16-bit FAT; partitions ≥32M)
7	OS/2 HPFS
7	Previous QNX version 2 (pre-1988)

continued...

Type	Filesystem
8	QNX 1.x and 2.x (“ qny ”)
9	QNX 1.x and 2.x (“ qnz ”)
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 POSIX partition (secondary)
78	QNX POSIX partition (secondary)
79	QNX POSIX partition
99	UNIX
131	Linux (Ext2)

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.

The following chart highlights some of the POSIX-defined capabilities and indicates if the filesystems support them:

Capability	Image	RAM	ETFS	QNX4	DOS	CD-ROM	FFS3	NFS	CIFS	Ext2
Access date	no	yes	yes	yes	yes ^a	yes ^b	no	yes	no	yes
Modification date	no	yes	yes	yes	yes	yes ^b	yes	yes	yes	yes
Status change date	no	yes	yes	yes	no	yes ^b	yes	yes	no	yes
Filename length	255	255	91	48 ^c	8.3 ^d	32 ^e	255	— ^f	— ^f	255
User permissions	yes	yes	yes	yes	no	yes ^b	yes	yes ^f	yes ^f	yes
Group permissions	yes	yes	yes	yes	no	yes ^b	yes	yes ^f	yes ^f	yes
Other permissions	yes	yes	yes	yes	no	yes ^b	yes	yes ^f	yes ^f	yes
Directories	no	no	yes	yes	yes	yes	yes	yes	yes	yes
Hard links	no	no	no	yes	no	no	no	yes ^f	no	yes
Soft links	no	no	yes	yes	no	yes ^b	yes	yes ^f	no	yes
Decompression on read	no	no	no	no	no	no	yes	no	no	no

^a VFAT or FAT32 (e.g. Windows 95).

^b With Rock Ridge extensions.

^c 505 if **.longfilenames** is enabled; otherwise, 48.

^d 255-character filename lengths used by VFAT or FAT32 (e.g. Windows 95).

^e 128 with Joliet extensions; 255 with Rock Ridge extensions.

^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 system also provides a simple RAM-based “filesystem” that allows read/write files to be placed under `/dev/shmem`.

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 utilizing 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 will be available shortly after the release of QNX Momentics 6.3.0. If you're interested in using this filesystem, please contact your QNX sales representative.

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.

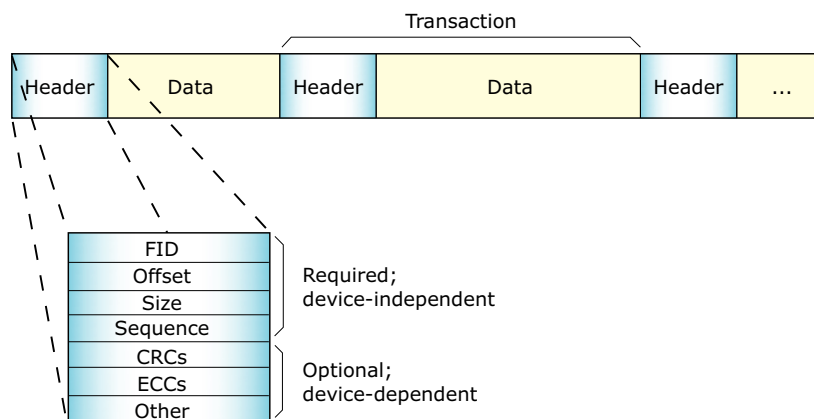
ETFS is a filesystem composed entirely of *transactions*. Every write operation, whether of user data or filesystem metadata, consists of a transaction. A transaction either succeeds or is treated as if it never occurred.

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.



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 through the use of driver classes as follows:

Class	CRC	ECC	Wear-leveling erase	Wear-leveling read	Cluster size
NAND 512+16	Yes	Yes	Yes	Yes	1K
NAND 2048+64	Yes	Yes	Yes	Yes	2K
RAM	No	No	No	No	1K
SRAM	Yes	No	No	No	1K
NOR	Yes	No	Yes	No	1K



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 under-worked static blocks and forces them into service by copying their data to an over-worked block. This solves two problems: It gives the over-worked block a rest, since it now contains static data, and it forces the under-worked 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 will mark the weak block for a *refresh* operation, which copies the data to a new flash block and erases the weak block. The erase 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 will recover 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 defrag 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.

QNX4 filesystem

The QNX4 filesystem (**fs-qnx4.so**) is a high-performance filesystem that shares the same on-disk structure as QNX 4.

The QNX4 filesystem implements an extremely robust design, utilizing an extent-based, bitmap allocation scheme with fingerprint control structures to safeguard against data loss and to provide easy recovery. Features include:

- extent-based POSIX filesystem
- robustness: all sensitive filesystem info is written through to disk
- on-disk “signatures” and special key information to allow fast data recovery in the event of disk damage
- 505-character filenames
- multi-threaded design
- client-driven priority
- same disk format as the filesystem under QNX 4.



Since the release of 6.2.1, the 48-character filename limit has increased to 505 characters via a backwards-compatible extension. The same on-disk format is retained, but new systems will see the longer name, old ones will see a truncated 48-character name.

For more information, see “QNX4 filesystem” in the Working With Filesystems chapter of the QNX Neutrino *User’s Guide*.

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 QNX 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** will 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 both floppies and 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 won't be 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 cannot 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 will be 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) won’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.

CD-ROM filesystem

The CD-ROM filesystem provides transparent access to CD-ROM media, so you can treat CD-ROM filesystems as though they were POSIX filesystems. This transparency allows processes to operate on CD-ROM files without any special knowledge or work on their part.

The **fs-cd.so** manager implements the ISO 9660 standard as well as a number of extensions, including Rock Ridge (RRIP), Joliet (Microsoft), and multisession (Kodak Photo CD, enhanced audio).

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. For example, the **devf-800fads** driver is for the 800FADS PowerPC evaluation board.

To find out what flash devices we currently support, please refer to the following sources:

- the **boards** and **mt-d-flash** directories under *bsp_working_dir/src/hardware/flash*.
- QNX Neutrino OS docs (**devf-*** entries in *Utilities Reference*).
- the QNX Software Systems website (www.qnx.com).

Customization

Along with the pre-built flash filesystem drivers, including the “generic” driver (**devf-generic**), provide the libraries and source code needed to build custom flash filesystem drivers for different embedded systems. For information on how to do this, see the Customizing the Flash Filesystem chapter in the *Building Embedded Systems* book.

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 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.

Mountpoints

When you start the flash filesystem driver, it will by default mount any partitions it finds in the socket. Note that you can specify the mountpoint using **mkefs** or **flashctl** (e.g. **/flash**).

Mountpoint	Description
<code>/dev/fsX</code>	raw mountpoint socket <i>X</i>
<code>/dev/fsXpY</code>	raw mountpoint socket <i>X</i> partition <i>Y</i>
<code>/fsXpY</code>	filesystem mountpoint socket <i>X</i> partition <i>Y</i>
<code>/fsXpY/.cmp</code>	filesystem compressed mountpoint socket <i>X</i> partition <i>Y</i>

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.

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/decompression

For fast and efficient compression/decompression, you can use the **deflate** and **inflater** 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 *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 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.

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 98 or NT 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** module uses TCP/IP for its transport. Therefore, to use **fs-cifs** (**SMBfsys** in QNX 4), you'll also need to run the TCP/IP client for Neutrino.

Linux Ext2 filesystem

The Ext2 filesystem (**fs-ext2.so**) provides transparent access to Linux disk partitions. This 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).

Virtual filesystems

QNX Neutrino provides two types of *virtual* filesystems:

- Package filesystem

- Inflator

Package filesystem

Our package filesystem component is a virtual filesystem manager that presents a customized view of a set of files and directories to a client. The actual files are present on some medium (e.g. a server's hard disk) — the package filesystem presents a “virtual” view of selected files to the client.

This can be useful in a networked environment, where a centralized server maintains a separate package of files for each client machine. Each client may want to see their own custom versions of certain files or unique combinations of files in a package.

Traditionally, you would achieve this either by copying all the files for all the packages into directories, with one directory per machine, or by creating various symbolic links.

By capitalizing on QNX Neutrino's pathname space and resource manager concepts, the package filesystem manager alleviates these problems. You can easily install a set of directories and files, and just as easily *uninstall* them, all without having to copy the files themselves.

What's in a package?

A package consists of a number of files and directories that are related to each other by virtue of being part of a product or of a particular release. For example, the QNX Neutrino OS, including its binary files and online documentation, is considered a package. A set of updates (called a *patch*) to the OS might contain only selected files and documentation (i.e. the files that have changed); this would be a different package.

The purpose of the package filesystem is to manage these packages in such a way that various nodes can pick and choose which particular packages they want to use. For example, one node might be an x86 box running, say, Patch B of a certain release, while another node might be a PowerPC box running Patch C.

Since these two machines are different CPU types, we need to give each of them not only different types of executables in their **/bin** directories (one set of x86 executables and one set of PowerPC executables), but also different contents based on the desired version of the software and patch levels.

Single definition file

In a nutshell, the package filesystem presents, on each node, a virtual filesystem that contains only selected files from the master package database. The selection is controlled by a *definition file* that indicates:

- which packages should be presented
- where the master package database resides
- where the virtual filesystem is to be rooted.

The advantage of a single definition file is clear: When the time comes to change the package lineup on a particular node, this can be accomplished by a simple (and *fast!*) change to the definition file.

Inflator

Inflator 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.



Chapter 9

Character I/O

In this chapter...

Introduction	215
Console devices	224
Serial devices	225
Parallel devices	225
Pseudo terminal devices (ptys)	226



Introduction

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:

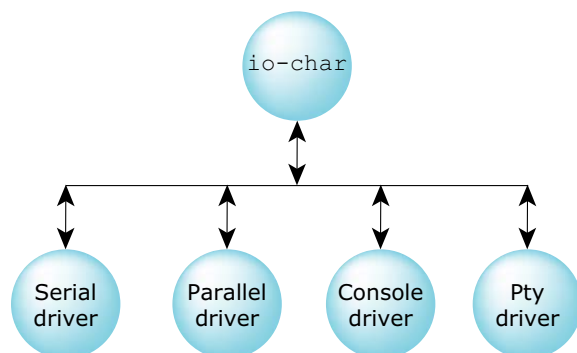
`/dev/ser1`

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.



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.

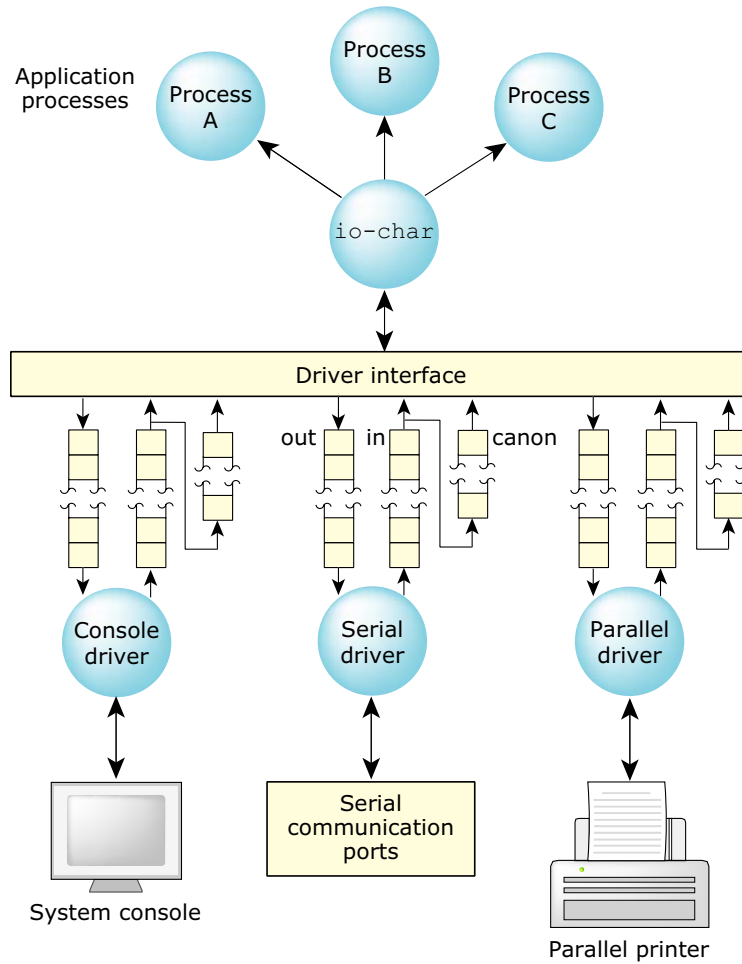
The driver is the executing process that calls into the library. In operation, the driver starts first and invokes `io-char`. The drivers themselves are just like any other QNX Neutrino process and can run at different priorities according to the nature of the hardware being controlled and the client's 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.



Device I/O in QNX Neutrino.

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:

<i>tcgetattr()</i>	Get terminal attributes.
<i>tcsetattr()</i>	Set terminal attributes.
<i>tcgetpgrp()</i>	Get ID of process group leader for a terminal.
<i>tcsetpgrp()</i>	Set ID of process group leader for a terminal.

<i>tcsendbreak()</i>	Send a break condition.
<i>tcflow()</i>	Suspend or restart data transmission/reception.

QNX extensions

The QNX extensions to the terminal control API are as follows:

<i>tcdropline()</i>	Initiate a disconnect. For a serial device, this will pulse the DTR line.
<i>tcinject()</i>	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.

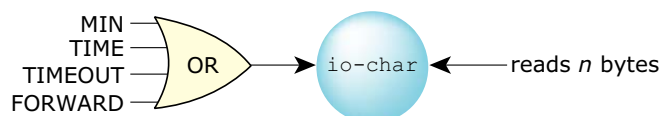
Raw input mode

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.

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 timeout a read if no response is available within a given time.

TIMEOUT is a QNX 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 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-tcon** driver. The video display card/screen and the system keyboard are collectively referred to as the *physical console*.

The **devc-con** 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.

devc-tcon, a “tiny” implementation of **devc-con**, is intended for memory-constrained systems and supports only a single console.

Terminal emulation

Both console drivers emulate an ANSI terminal; **devc-tcon** implements a subset of **devc-con**'s extended ANSI emulation.

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 has several serial drivers (e.g. **devc-ser8250**, **devc-serppc800**, etc.). For details, see the (**devc-ser*** entries in *Utilities Reference*).



Many drivers have a “tiny” version (**devc-t***) intended for memory-constrained systems. Note that the tiny versions support raw mode only.

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).

Parallel devices

Parallel printer ports are managed by the **devc-par** driver. When **devc-par** is started, command-line arguments can specify which parallel port is installed.

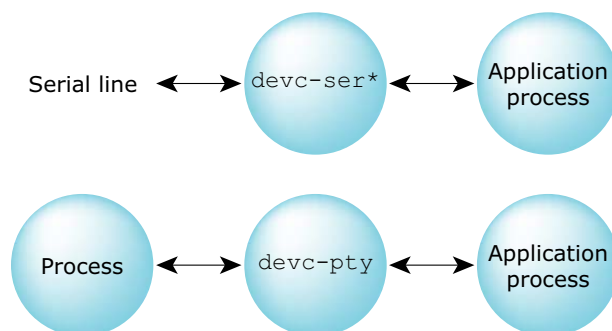
The **devc-par** driver is an output-only driver, so it has no raw input or canonical input queues. The size of the output buffer can be

configured with a command-line argument. If configured to a large size, this creates the effect of a software print buffer.

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.



Pseudo-ttys.

Ptys are routinely used to create pseudo-terminal interfaces for programs like **pterm**, a terminal emulator that runs under the Photon microGUI and **telnet**, which uses TCP/IP to provide a terminal session to a remote system.

Chapter 10

Networking Architecture

In this chapter...

Introduction	229
Network manager (io-net)	230
Filter module	231
Converter module	231
Protocol module	231
Driver module	232



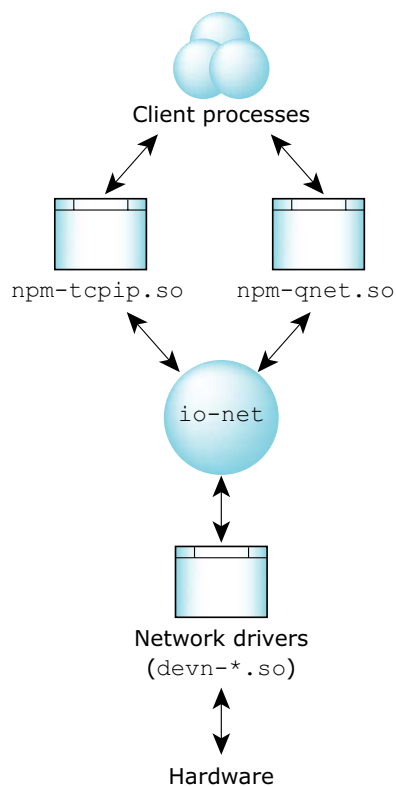
Introduction

As with other service-providing processes in QNX Neutrino, 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, TCP/IP, and other protocols to run together in any combination.

Our native network subsystem consists of the network manager executable (**io-net**), plus one or more shared library modules. These modules can include protocols (e.g. **npm-qnet.so**, **npm-tcpip.so**), drivers (e.g. **devn-ne2000.so**), and filters.



*The **io-net** process.*

Network manager (**io-net**)

The **io-net** component is the active executable within the network subsystem. Acting as a kind of packet redirector/multiplexer, **io-net** 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-net** executable efficiently loads multiple networking protocols, filters, or drivers (e.g. **npm-qnet.so**, **npm-tcpip.so**) *on the fly* — these modules are shared objects that install into **io-net**.

Filter module

The **io-net** framework lets you set up a filter module, which can be registered above or below a producer module, such as a driver or protocol module.

A filter module allows you to intercept data packets as they're passed from this producer **io-net** module. This allows you to modify, drop, or simply monitor these packets. You can also direct packets to other interfaces (e.g. bridge or forward). Typically, a filter module would be registered above a network driver module (e.g. Ethernet).

Converter module

The basic role of the converter module is to encapsulate and de-encapsulate packets as they pass from one network layer to another (e.g. IP to Ethernet). You use converters to connect producer modules together (e.g. a network protocol stack to a network driver).

These modules may also implement the protocols used to resolve the addressing used by the network protocol module to the physical network addresses supported by the network driver. For example, the ARP protocol (IP-to-Ethernet address translation) could be implemented as part of a converter module.

Protocol module

The networking protocol module is responsible for implementing the details of a particular protocol (e.g. Qnet, TCP/IP, etc.). Each protocol component is packaged as a shared object (e.g. **npm-qnet.so**). One or more protocol components may run concurrently.

For example, the following line from a buildfile shows **io-net** loading two protocols (TCP/IP and Qnet) via its **-p protocol** command-line option:

```
io-net -dne2000 -ptcpip -pqnet
```

- | | |
|--------------|--|
| tcpip | We've implemented the BSD TCP/IP stack as a protocol module. |
| qnet | <p>Qnet is the QNX Neutrino native networking protocol. Its main purpose is to extend the OS's powerful message-passing IPC <i>transparently</i> over a network of microkernels.</p> <p>Qnet also provides Quality of Service policies to help ensure reliable network transactions.</p> |

For more information on the Qnet and TCP/IP protocols, see the following chapters in this book:

- Qnet Networking
- 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-net** component.

Loading and unloading a driver

Once **io-net** is running, you can dynamically load drivers at the command line using the **mount** command. For example:

```
io-net &  
mount -T io-net devn-ne2000.so
```

would start **io-net** and then mount the driver for an NE-2000 Ethernet adapter. All network device drivers are shared objects of the form:

```
devn-driver.so
```

Once the shared object is loaded, **io-net** will then initialize it. The driver and **io-net** are then effectively bound together — the driver will call into **io-net** (for example when packets arrive from the interface) and **io-net** will call into the driver (for example when packets need to be sent from an application to the interface).

You can also use the **umount** command to unload a driver:

```
umount /dev/io-net/en0
```

For more information on network device drivers, see their individual utility pages (**devn-***) in the *Utilities Reference*.

Network DDK

Although several network drivers are shipped with the OS, you may want to write your own driver for your particular networking hardware. The Network Driver Development Kit makes this task relatively easy. The DDK provides full source code for several sample drivers as well as detailed instructions for handling the hardware-dependent issues involved in developing custom drivers for the **io-net** infrastructure.



Chapter 11

Native Networking (Qnet)

In this chapter...

QNX Neutrino distributed 237
Name resolution and lookup 239
Quality of Service (QoS) and multiple paths 245
Examples 249
Custom device drivers 251



QNX Neutrino distributed

Earlier in this manual, we described message passing in the context of a single node (see the section “QNX Neutrino IPC” in the microkernel chapter). But the true power of QNX Neutrino 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). For information on TCP/IP networking, please refer to the next chapter.

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 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.

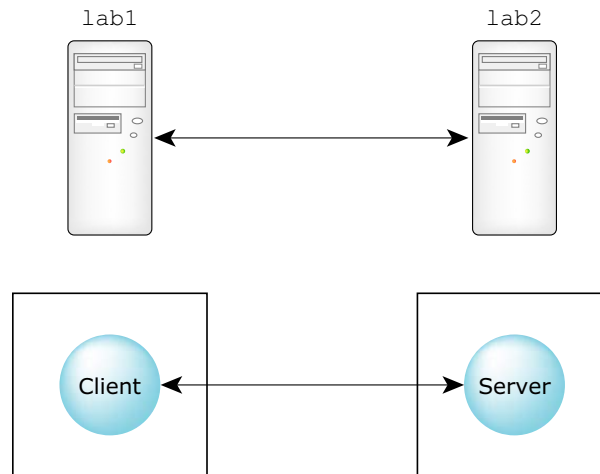
- Divide your large application into several processes, where each process can perform different functions. These processes coordinate their work using message passing.
- Take advantage of Qnet's inherent remote procedure call functionality.

Moreover, since Qnet extends 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.



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 Networking via Qnet chapter of the *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, `npm-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

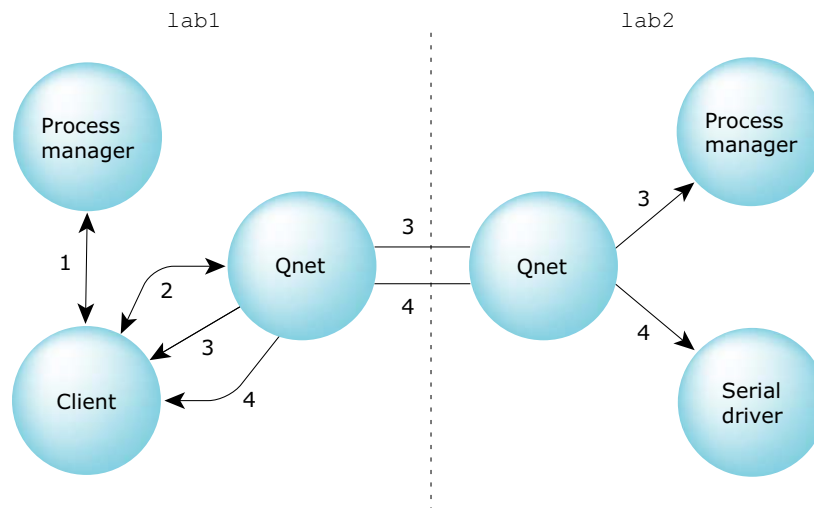
File descriptor (connection ID)

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**:



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/1ab2/dev/ser1`.

Since the native network manager (`npm-qnet`) 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 `1ab2` — effectively deferring the resolution of the request to node `1ab2`.
- 3** The client then creates a connection to the process manager on node `1ab2`, once again asking who should be contacted to resolve the pathname.

The process manager on node `1ab2` 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 `1ab2`, 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 automatically 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 **npm-qnet** uses. In resolving names in a network-wide pathname space, the following terms come into play:

<i>node name</i>	A character string that identifies the node you're talking to. Note that a node name <i>can't</i> contain slashes or dots. In the example above, we used lab2 as one of our node names. The default is fetched via <i>confstr()</i> with the _CS_HOSTNAME parameter.
------------------	--

node domain A character string that's "tacked" onto the node name by **npm-qnet**. 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 **npm-qnet**. 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 **npm-qnet** 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 **ndp** (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 *resolvers* are built into the network manager:

- **ndp** — Node Discovery Protocol for broadcasting name resolution requests 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.

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 the following policies for specifying how Qnet should select a network interface for transmission:

loadbalance (the default)

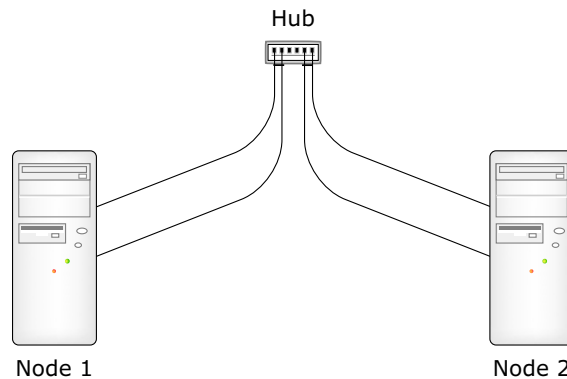
Qnet is free to use all available network links, and will share transmission equally among them.

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.

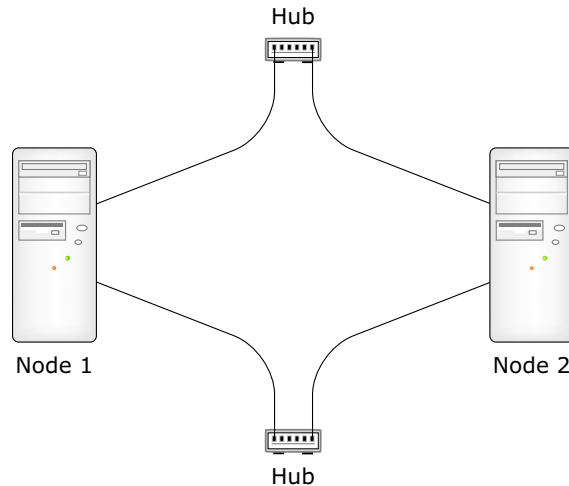
To fully benefit from Qnet's QoS, you need to have physically separate networks. For example, consider a network with two nodes and a hub, where each node has two connections to the hub:



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:



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.

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-net**. 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. (This is a significant improvement over the QNX 4 implementation.)

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.



The **loadbalance** QoS policy is the default.

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.

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 **npm-qnet** is actually a shared object that installs into the executable **io-net**.

Local networks

If you're using QNX Neutrino on a small LAN, you can use just the default **ndp** 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 **ndp** is the default resolver when you start **npm-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



CAUTION: For security reasons, you should have a firewall set up on your network before connecting to the Internet. For more information, see the section “Setting up a firewall” in the chapter *Securing Your System* in the *User's Guide*.

Qnet uses DNS (Domain Name System) when resolving remote names. To use **npm-qnet.so** with DNS, you specify this resolver on **mount**'s command line:

```
mount -Tio-net -o"mount=,resolve=dns,mount=.com:.net:.edu" /lib/dll/npm-qnet.so
```

In this example, Qnet will use *both* its native **ndp** 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.

Custom device drivers

In most cases, you can use standard QNX drivers to implement Qnet over a local area network or to encapsulate Qnet messages in IP (TCP/IP) to allow Qnet to be routed to remote networks. But suppose you want to set up a very tightly coupled network between two CPUs over a super-fast interconnect (e.g. PCI or RapidIO)?

You can easily take advantage of the performance of such a high-speed link, because Qnet can talk directly to your hardware driver. There's no **io-net** layer in this case. All you need is a little code at the very bottom of the Qnet layer that understands how to transmit and receive packets. This is simple, thanks to a standard internal API between the rest of Qnet and this very bottom portion, the driver interface.

Qnet already supports different packet transmit/receive interfaces, so adding another is reasonably straightforward. Qnet's transport mechanism (called "L4") is quite generic, and can be configured for different size MTUs, for whether or not ACK packets or CRC checks are required, etc., to take full advantage of your link's advanced features (e.g. guaranteed reliability).

A Qnet software development kit is available to help you write custom drivers and/or modify Qnet components to suit your particular application.



Chapter 12

TCP/IP Networking

In this chapter...

Introduction	255
Structure of TCP/IP manager	257
Socket API	258
Multiple stacks	260
SCTP	260
IP filtering and NAT	261
NTP	262
Dynamic host configuration	262
PPP over Ethernet	263
<code>/etc/autoconnect</code>	263
SNMP support	264
Embedded web server	264



Introduction

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. QNX Neutrino 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 (**`npm-tcpip`**) relatively light on resources, while using the common BSD API.

Stack configurations

We provide three stack configurations:

NetBSD TCP/IP stack

Supports the latest RFCs, including UDP, IP, TCP, and SCTP. 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.

Tiny TCP/IP stack

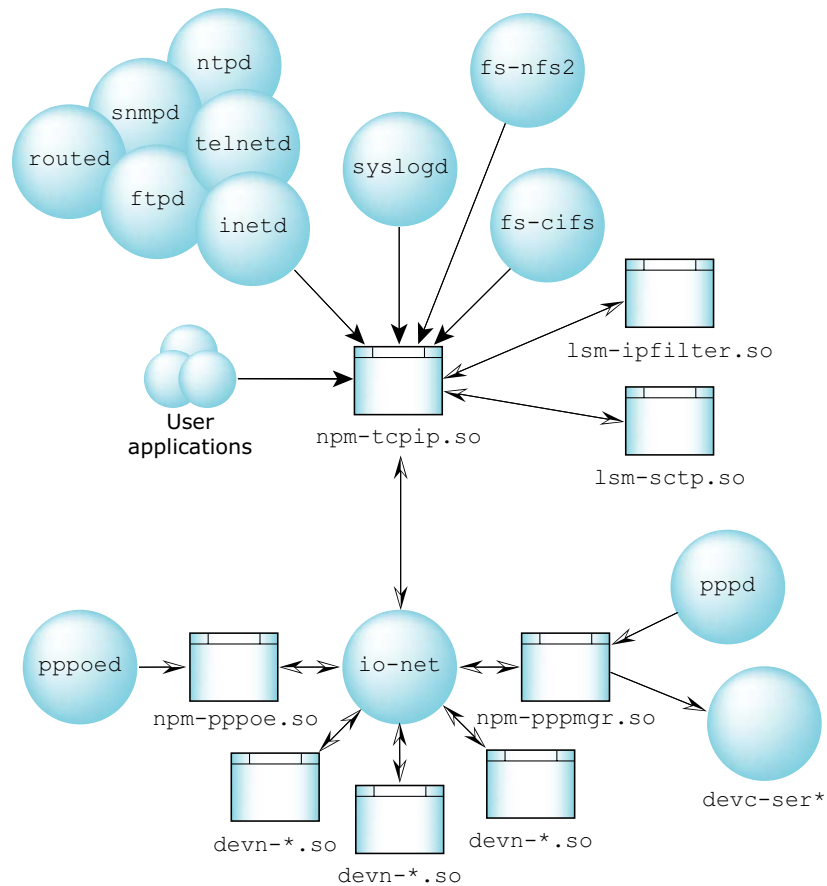
Designed for memory-constrained systems, this small-footprint (80K) stack provides complete support for IP, TCP, and UDP over Ethernet, PPP, and PPPoE interfaces. To develop applications, you use the BSD socket interface; in fact, you can switch between the tiny stack and NetBSD stacks without having to recompile your code.

The QNX 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.

To customize our TCP/IP suite, developers can also obtain a source release of several components (e.g. **io-net**, **npm-tcpip.so**). The source kit also includes simple source examples of **io-net** modules to help developers create or port source code for the **io-net** architecture.

Structure of TCP/IP manager

As a resource manager, the **npm-tcpip** module 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.



*The **npm-tcpip** suite and depends.*

Most of the link-layer support is implemented outside of **npm-tcpip**. PPP is implemented as separate shared objects. Since our **io-net** PPP modules (**npm-pppmgr.so**, **npm-pppoe.so**) handle 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.

Note that **npm-tcpip** acts as a protocol shared library to the **io-net** process, which directs data to the correct driver. Network drivers are shared libraries as well.

Other components of the **npm-tcpip** suite (such as the NFS, the **snmpd** daemon, etc.) are implemented outside of **npm-tcpip**. This leads to better modularity and fault-tolerance.

Socket API

The BSD Socket API was the obvious choice for QNX Neutrino. 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):

<i>accept()</i>	<i>getprotoent()</i>
<i>bind()</i>	<i>getservbyname()</i>
<i>bindresvport()</i>	<i>getservent()</i>
<i>connect()</i>	<i>getsockname()</i>
<i>dn_comp()</i>	<i>getsockopt()</i>
<i>dn_expand()</i>	<i>herror()</i>
<i>endprotoent()</i>	<i>hstrerror()</i>
<i>endservent()</i>	<i>htonl()</i>
<i>gethostbyaddr()</i>	<i>htons()</i>
<i>gethostbyname()</i>	<i>h_errlist()</i>
<i>getpeername()</i>	<i>h_errno()</i>
<i>getprotobyname()</i>	<i>h_nerr()</i>
<i>getprotobynumber()</i>	<i>inet_addr()</i>

<i>inet_aton()</i>	<i>res_mkquery()</i>
<i>inet_lnaof()</i>	<i>res_query()</i>
<i>inet_makeaddr()</i>	<i>res_querydomain()</i>
<i>inet_netof()</i>	<i>res_search()</i>
<i>inet_network()</i>	<i>res_send()</i>
<i>inet_ntoa()</i>	<i>select()</i>
<i>ioctl()</i>	<i>send()</i>
<i>listen()</i>	<i>sendto()</i>
<i>ntohl()</i>	<i>setprotoent()</i>
<i>ntohs()</i>	<i>setservent()</i>
<i>recv()</i>	<i>setsockopt()</i>
<i>recvfrom()</i>	<i>shutdown()</i>
<i>res_init()</i>	<i>socket()</i>

For more information, see the *Neutrino 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 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-net**) lets you load *multiple* protocol shared objects (e.g. **npm-tcpip.so**). This means, for example, that you can load several instances of the TCP/IP stack *on the same physical interface*, making it easy to create multiple virtual networks (VLANs). You can even run multiple, independent instances of the network manager (**io-net**) itself. As with all QNX Neutrino system components, each **io-net** naturally benefits from complete memory protection thanks to our microkernel architecture.

SCTP

The SCTP (Stream Control Transport Protocol) module (**lsm-sctp.so**) is a dynamically loadable module that adds SCTP protocol processing to your TCP/IP stack. SCTP is a reliable connection-oriented transport protocol with features such as:

- acknowledged error-free, non-duplicated transfer of user data
- data fragmentation to conform to the discovered path MTU size
- sequenced delivery of user messages within multiple streams
- multi-homing.

The QNX SCTP implementation makes use of the BSD socket-based API routines along with SCTP-specific API extensions, including:

- `sctp_peeloff()`
- `sctp_bindx()`

- *sctp_connectx()*
- *sctp_getpaddrs()*
- *sctp_freepaddrs()*
- *sctp_getladdrs()*
- *sctp_freeladdrs()*
- *sctp_opt_info()*
- *sctp_sendmsg()*
- *sctp_rcvmsg()*

IP filtering and NAT

The IP filtering and NAT (Network Address Translation) **io-net** module (**lsm-ipfilter-*.so**) is a dynamically loadable TCP/IP stack module. This 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 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 (**dhcp.client**) 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 **dhcp.client**, **dhcpd**, and **dhcrelay** entries in the *Utilities Reference*.

AutoIP

Developed from the Zeroconf IETF draft, **nfm-autoip.so** is an **io-net** 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 (**dhcp.client**), 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 **npm-pppoe.so** shared object as well as the **pppoed** daemon, 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 everything up for you.

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

npm-pppoe.so

Shared object that provides PPP-to-Ethernet services.

pppoed

Daemon to negotiate the PPPoE session.

phlip

Photon TCP/IP and dialup configuration tool.

phdialer

Photon modem dialer.

/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.

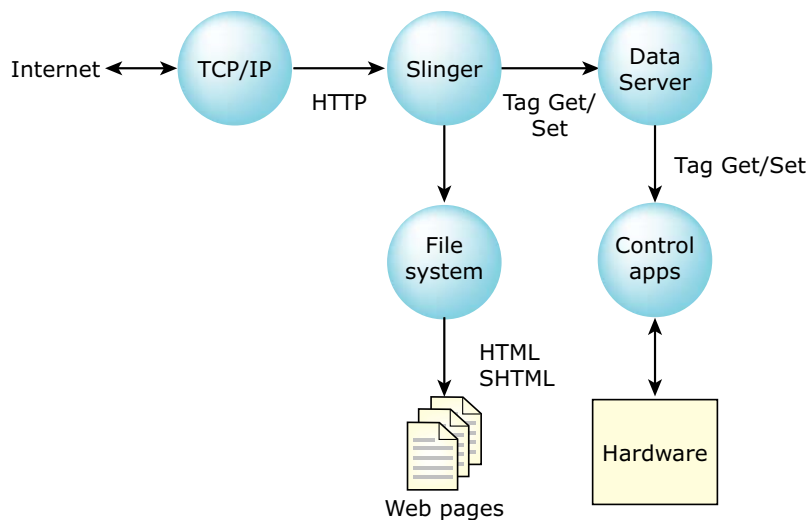
For more information, see the entry for **/etc/autoconnect** in the *Utilities Reference*.

SNMP support

The SRI SNMP Suite for QNX Neutrino consists primarily of ports of the EMANATE and EMANATE/Lite technologies developed by SNMP Research International (SRI). EMANATE/Lite is a statically extensible agent; the EMANATE agent can be extended dynamically. Both agents support SNMP V1, V2, and V3, and include development kits for developers to extend the agents.

Embedded web server

Our embedded web server, **slinger**, uses very little memory and communicates over TCP/IP sockets. The embedded web server supports CGI 1.1, HTTP 1.1, and dynamic HTML (via SSI commands).



Embedded web server.

Many embedded servers force the user to relink the server in order to add pages, which compromises reliability as vendor and user code compete in a shared memory space.

Despite its size, our embedded web server provides enough functionality to support accessing generated (dynamic) HTML via CGI or SSI.

CGI method

The embedded web server supports the Common Gateway Interface (CGI) 1.1, a readily available means of handling dynamic data. The downside of CGI is that it's resource-heavy because an interpreted language is often involved.

SSI method

With SSI (Server Side Includes), a type of command language that can be embedded in HTML files, you can add dynamic content to your HTML. For example, the embedded server can:

- execute utilities at user-defined points in an HTML document (the output of these utilities can be optionally placed in the document).
- insert contents of other HTML files at a user-defined point.
- handle conditional statements (**if**, **break**, **goto**) so you can define what parts of an HTML file are transmitted.

Note that SSI tags are available to interact with a data server.

Data server method

You can also handle dynamic HTML by using what we call a *data server*. The data server allows multiple threads to share data without regard for process boundaries. Since the embedded web server supports SSI, we've extended this support by adding the ability to talk to the data server.

Now you can have a process updating the data server about the state of a hardware device while the embedded web server accesses that state in a decoupled but reliable manner.



You can write a simple I/O manager to provide dynamic data. Note that this method isn't specific to the embedded web server and that the I/O manager can handle only output, not posts.

Chapter 13

High Availability

In this chapter...

What is High Availability?	269
Client library	271
High Availability Manager	274



What is 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 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, 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 HA 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 file system. 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-nfs2 fails, and is restarted, the network mounts
    // are re-instated 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 points 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

HAM 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 *pids*. 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.

continued...

Condition	Description
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.
CONDBEATMISSEDHIGH	The entity missed sending a "heartbeat" message specified for a condition of "high" severity.
CONDBEATMISSEDLow	The entity missed sending a "heartbeat" message specified for a condition of "low"
CONDRESTART	The entity was restarted. This condition is true <i>after</i> 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.

continued...

Condition	Description
CONDANY	This condition type matches <i>any</i> 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
<i>ham_action_restart()</i>	This action restarts the entity.
<i>ham_action_execute()</i>	Executes an arbitrary command (e.g. to start a process).

continued...

Action	Description
<i>ham_action_notify_pulse()</i>	Notifies some process that this condition has occurred. This notification is sent using a specific <i>pulse</i> with a value specified by the process that wished to receive this notify message.
<i>ham_action_notify_signal()</i>	Notifies some process that this condition has occurred. This notification is sent using a specific <i>realtime signal</i> with a value specified by the process that wished to receive this notify message.
<i>ham_action_notify_pulse_node()</i>	This is the same as <i>ham_action_notify_pulse()</i> above, except that the node name specified for the recipient of the pulse can be the fully qualified node name.
<i>ham_action_notify_signal_node()</i>	This is the same as <i>ham_action_notify_signal()</i> above, except that the node name specified for the recipient of the signal can be the fully qualified node name.
<i>ham_action_waitfor()</i>	Lets you insert delays between consecutive actions in a sequence. You can also wait for certain names to appear in the namespace.

continued...

Action	Description
<i>ham_action_heartbeat_healthy()</i>	Resets the heartbeat mechanism for an entity that had previously missed sending heartbeats and had triggered a missed heartbeat condition, but has now recovered.
<i>ham_action_log()</i>	Reports this condition to a logging mechanism.

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

Alternate actions

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 API call. For more information, see the API reference documentation for the *ham_condition_state()* call in the High Availability Toolkit *Developer's Guide*.

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, refer to the API documentation for the *ham_condition_raise()* call in the High Availability Toolkit *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-nfs2` (the NFS filesystem) and then mounted a few directories from multiple sources. You can instruct HAM to restart `fs-nfs2` upon failure, and also to remount the appropriate directories as required after restarting the NFS process.

As another example, suppose `io-net` (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
<i>ham_action_control()</i>	Perform control operations on an action object.
<i>ham_action_execute()</i>	Add an execute action to a condition.
<i>ham_action_fail_execute()</i>	Add to an action an execute action that will be executed if the corresponding action fails.
<i>ham_action_fail_log()</i>	Insert a log message into the activity log.
<i>ham_action_fail_notify_pulse()</i>	Add to an action a notify pulse action that will be executed if the corresponding action fails.
<i>ham_action_fail_notify_pulse_node()</i>	Add to an action a node-specific notify pulse action that will be executed if the corresponding action fails.
<i>ham_action_fail_notify_signal()</i>	Add to an action a notify signal action that will be executed if the corresponding action fails.
<i>ham_action_fail_notify_signal_node()</i>	Add to an action a node-specific notify signal action that will be executed if the corresponding action fails.

continued...

Function	Description
<i>ham_action_fail_waitfor()</i>	Add to an action a waitfor action that will be executed if the corresponding action fails.
<i>ham_action_handle()</i>	Get a handle to an action in a condition in an entity.
<i>ham_action_handle_node()</i>	Get a handle to an action in a condition in an entity, using a nodename.
<i>ham_action_handle_free()</i>	Free a previously obtained handle to an action in a condition in an entity.
<i>ham_action_heartbeat_healthy()</i>	Reset a heartbeat's state to healthy.
<i>ham_action_log()</i>	Insert a log message into the activity log.
<i>ham_action_notify_pulse()</i>	Add a notify-pulse action to a condition.
<i>ham_action_notify_pulse_node()</i>	Add a notify-pulse action to a condition, using a nodename.
<i>ham_action_notify_signal()</i>	Add a notify-signal action to a condition.
<i>ham_action_notify_signal_node()</i>	Add a notify-signal action to a condition, using a nodename.
<i>ham_action_remove()</i>	Remove an action from a condition.

continued...

Function	Description
<i>ham_action_restart()</i>	Add a restart action to a condition.
<i>ham_action_waitfor()</i>	Add a waitfor action to a condition.
<i>ham_attach()</i>	Attach an entity.
<i>ham_attach_node()</i>	Attach an entity, using a nodename.
<i>ham_attach_self()</i>	Attach an application as a self-attached entity.
<i>ham_condition()</i>	Set up a condition to be triggered when a certain event occurs.
<i>ham_condition_control()</i>	Perform control operations on a condition object.
<i>ham_condition_handle()</i>	Get a handle to a condition in an entity.
<i>ham_condition_handle_node()</i>	Get a handle to a condition in an entity, using a nodename.
<i>ham_condition_handle_free()</i>	Free a previously obtained handle to a condition in an entity.
<i>ham_condition_raise()</i>	Attach a condition associated with a condition raise condition that's triggered by an entity raising a condition.
<i>ham_condition_remove()</i>	Remove a condition from an entity.

continued...

Function	Description
<i>ham_condition_state()</i>	Attach a condition associated with a state transition condition that's triggered by an entity reporting a state change.
<i>ham_connect()</i>	Connect to a HAM.
<i>ham_connect_nd()</i>	Connect to a remote HAM.
<i>ham_connect_node()</i>	Connect to a remote HAM, using a nodename.
<i>ham_detach()</i>	Detach an entity from a HAM.
<i>ham_detach_name()</i>	Detach an entity from a HAM, using an entity name.
<i>ham_detach_name_node()</i>	Detach an entity from a HAM, using an entity name and a nodename.
<i>ham_detach_self()</i>	Detach a self-attached entity from a HAM.
<i>ham_disconnect()</i>	Disconnect from a HAM.
<i>ham_disconnect_nd()</i>	Disconnect from a remote HAM.
<i>ham_disconnect_node()</i>	Disconnect from a remote HAM, using a nodename.
<i>ham_entity()</i>	Create entity placeholder objects in a HAM.
<i>ham_entity_condition_raise()</i>	Raise a condition.

continued...

Function	Description
<i>ham_entity_condition_state()</i>	Notify the HAM of a state transition.
<i>ham_entity_control()</i>	Perform control operations on an entity object in a HAM.
<i>ham_entity_handle()</i>	Get a handle to an entity.
<i>ham_entity_handle_node()</i>	Get a handle to an entity, using a nodename.
<i>ham_entity_handle_free()</i>	Free a previously obtained handle to an entity.
<i>ham_entity_node()</i>	Create entity placeholder objects in a HAM, using a nodename.
<i>ham_heartbeat()</i>	Send a heartbeat to a HAM.
<i>ham_stop()</i>	Stop a HAM.
<i>ham_stop_nd()</i>	Stop a remote HAM.
<i>ham_stop_node()</i>	Stop a remote HAM, using a nodename.
<i>ham_verbose()</i>	Modify the verbosity of a HAM.

Chapter 14

Power Management

In this chapter...

Power management for embedded systems 291
Application-driven framework 291



Power management for embedded systems

Traditional power management (PM) is aimed at conserving the power of computers that are usually left *on*. The general-purpose approach to PM for desktop PCs — or even for “mobile” PCs, such as laptops — doesn’t take into account the specific demands of embedded systems, which can be *off* (or on standby) much of the time, yet must respond to external events in predictable ways.

The two main industry standards for PM — APM and its successor ACPI — deal with PCs, not embedded systems. The older APM standard was BIOS-oriented (how many embedded systems even have a BIOS?). The newer ACPI is more OS-focused and therefore more sophisticated, but the standard expressly concerns itself with desktops, laptops, and servers. An OS can never anticipate the application-specific scenarios of embedded systems and make appropriate power policy decisions.

Application-driven framework

Given the sheer variety of embedded systems (e.g. hand-held medical instruments, network routers, in-car telematics systems, to name but a few), the application developer is in the best position to know the specific power-related needs of the system in question.

The QNX PM framework is precisely that — a *framework* or set of mechanisms that developers can use to control and manage power in their embedded systems. The OS doesn’t impose a power policy. Instead, you create your own policy based on the specific needs of your application.

Framework components

The PM framework consists of these main components:

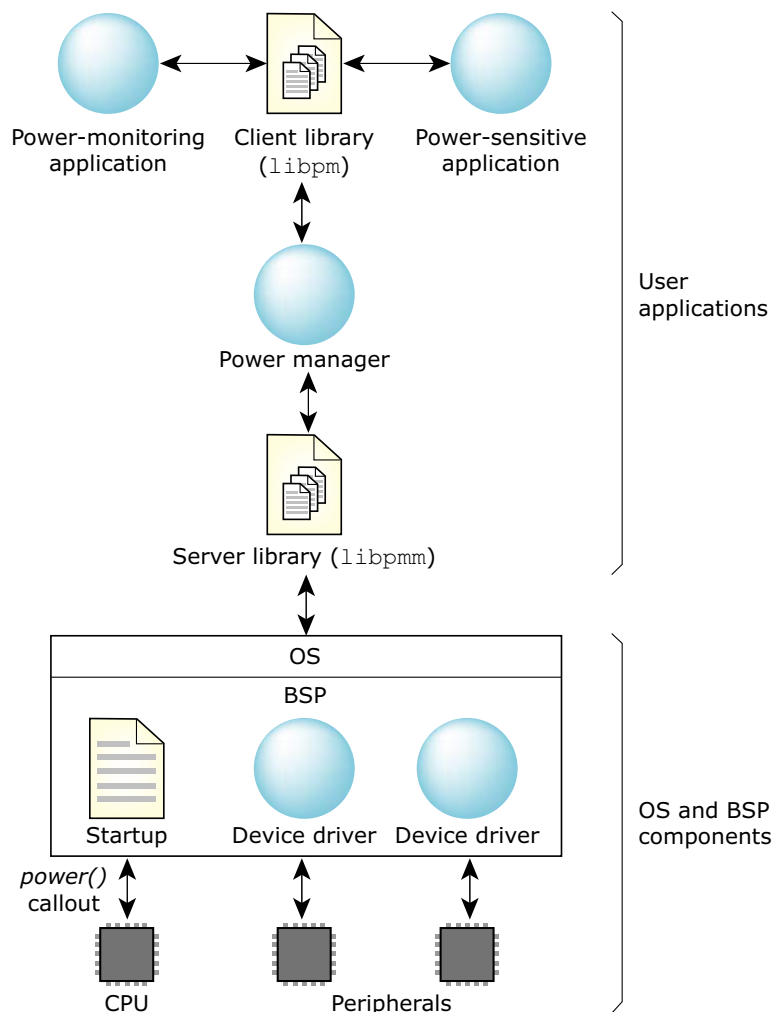
Framework component	Description
Power manager	A process that implements the system's PM policy. This policy is user-written — the system designer creates a custom policy for the embedded system, and has full control over device power modes for the various operational modes of the system.
Power-managed objects	These represent hardware or software components that can be independently power-managed. Device drivers for power-managed devices interact with the power manager to control device power modes according to the system's PM policy.
Power-aware applications	These are user-written applications that interact with the power manager. They provide functionality specific to the embedded system (e.g. monitoring environmental conditions that could affect the desired operational mode of the system).
Persistent storage services	These allow device drivers to preserve hardware and software configuration details when the system is placed in a standby mode. The developer can customize the physical storage location of this persistent data for the embedded system.
BSP support	Code within the board-specific IPL and startup programs implements the board- and CPU-specific support for sleep/standby modes.

Libraries and BSPs

These components are built using the following:

libpm	Client library that applications and device drivers use to interact with the power manager.
libps	Persistent storage library used to manipulate data managed by the persistent storage services.

- libpmm** Server library used to build the power manager. This library implements the underlying protocol for communicating with device drivers and client applications.
- BSPs** These provide reference implementations for the CPU power-mode support for reference hardware platforms (e.g. the Renesas Biscayne SH7760). You can customize a reference implementation for your particular embedded system.



The QNX PM framework facilitates application-specific designs.

Power manager

As the central component within the PM framework, the power manager is responsible for coordinating the power modes of each power-managed object according to the system's PM policy.

The power manager is implemented as a user-written process that uses the **libpmm** library to manage the interactions with device drivers and power-aware applications.

The power manager consists of three main components:

- A *resource manager interface* used to represent power-managed objects. Device drivers and power-aware applications use this interface to manipulate the objects controlled by the power manager.

The functionality for handling this communication is implemented entirely within the **libpmm** library.

- A *policy* that governs the behavior of power-managed objects in response to requests made via the resource manager interface.

The policy is customizable. Your policy is informed whenever an operation is performed on an object. This allows your policy code to track the state of an object, and where appropriate, to allow or deny certain operations. For example, your policy may deny requests to change the power mode of an object if the new mode isn't compatible with the system's current power-mode state.

The **libpm** library provides basic default behavior for these operations, so the system-specific policy needs to override only those operations that are required to implement the system's PM policy.

- *Code to manage system power states.* This user-written code defines the range of operational modes for the system along with their corresponding device and CPU power modes. This code is responsible for system initialization and actions that need to be performed based on system-specific events that are relevant to deciding the system's optimum power-mode state.

Resource manager interface

The most basic power manager simply initializes the resource manager interface to handle driver and client requests, and then starts a thread pool to service them:

```
#include <sys/procmgr.h>
#include <sys/pmm.h>

int
main()
{
    // initialise and start resource manager interface
    pmm_init(0, 0);
    pmm_start(0, 0);

    // become a daemon
    procmgr_daemon(0, 0);
    pthread_detach(pthread_self());
    pthread_exit(0);
    return 0;
}
```

This is sufficient to completely handle dynamic attachment by any drivers that start after the power manager; it implements a simple default policy that allows all power-mode changes to proceed as requested.

Power manager policy

The `libpmm` library handles the low-level details of interacting with device drivers and power-aware applications. However, the decisions about what actions are allowed to proceed are *policy decisions* that developers customize for their embedded system.

The library invokes this policy code whenever requests arrive that affect a power-managed object's state. If you don't supply your own policy function, an internal default is used.

The policy is informed (by invoking policy callback functions) when:

- An object is first created, and when it's deleted. This allows the policy to allocate its own specific data structures to be associated with the object.
- A client process attaches and detaches from an object. The policy can thus identify the client process and allow or reject access to the object. Since objects are accessed via a resource manager interface, basic authentication/access control is performed using the standard file access permissions of the object within the power

manager namespace. Customizing this callback can allow additional authentication mechanisms.

- An object is assigned a name in the namespace. This allows the policy to determine how this object is related to other objects in the namespace, for example to manage power dependencies. As with the client attachment, basic permission checking is provided via file access permissions.
- An object is removed from the namespace. This allows the policy to remove any inter-object dependencies.
- A driver asks to manage an object. The policy can thus discover the supported device modes and can specify what initial mode the device should be placed in. For warm reboots from system standby modes, the policy can also instruct the driver to recover any data it saved in persistent storage.
- A driver detaches. This informs the policy that the object can no longer be power-managed.
- A client (or the driver itself) requests a power-mode change to an object. The policy can validate that the requested mode is compatible with the current system power state and allow or reject the request.
- A property is added to an object, or a property value is changed. This allows the policy to react to property changes that may affect the system's power-mode state.

System power states

An embedded system may pass through a number of different power states. Each state defines the functionality and allowed power modes for the various power-managed objects in the system.

For example, an automotive device may define the following system states:

System state	Description
ACTIVE	A fully operational state where all devices may be powered up. Within this state, individual devices can be powered down when not being actively used, but are allowed to be powered up when required.
CRANK	A nonoperational state where the system is powered down during an engine crank. When the engine has started, the system returns to an active state.
SLEEP	A nonoperational state where devices are powered down and system RAM is held in self-refresh mode. This implements a “soft off” state that allows the system to become fully operational with short latency. Certain devices are configured to act as wakeup sources. For example, an RTC may be used to limit the time spent in this state. If the RTC timeout expires, the system will move to a lower power-standby state to further reduce power consumption.
STANDBY	A nonoperational state where devices are powered down and system RAM is disabled. This provides a lower-power “soft off” state, where the system can return to its previous operational state with a slightly longer latency, since it requires a warm reboot. Certain devices are configured to act as wakeup sources. For example, activating a key lock or phone interface would return the system to an active state, or an RTC can be used to periodically wake the system to ensure the system shuts down completely when the battery level reaches a certain threshold.

continued...

System state	Description
OFF	The entire system is powered down to prevent its consuming any battery power.

These system states can typically be modeled using a state machine, where system-specific events are used to cause transitions between states:

- Externally generated events can be implemented using properties (e.g. bus control messages associated with a vehicle bus, or discrete events such as key/switch status).
- Wakeup events are provided by the BSP-level IPL/startup code.

The policy callbacks interact with the state machine implementation to limit the allowed operations for the current system state, and to keep track of the status of each power-managed object so that they can be appropriately managed during state transitions.

Power-managed objects

Each entity that can be independently power-managed is represented by the power manager as a power-managed object.

The power manager's *system PM policy* determines the allowable power mode for each object, based on the system's current operational state. For example:

- System standby states require all power-managed objects to be placed in an appropriate standby mode.
- Some system states may operate with reduced power, providing a limited subset of the full system functionality. In this case, some devices are allowed to be powered up, while others must be powered down.

Power-managed objects are typically used to represent power-managed peripheral devices, allowing the power manager to control the device's power level. However, power-managed objects

can also represent specialized applications. In this case, their operational characteristics are exposed to the power manager via its power mode; the power manager can control the application's behavior by changing its mode when the power manager changes the system's power-mode state.

Power manager namespace

The power manager presents all power-managed objects via a hierarchical namespace under `/dev/pmm`. This namespace provides a simple configuration database that can represent both physical and logical relationships between objects:

- The hierarchy can follow the hardware topology between buses and devices. Nonleaf nodes represent buses, whose children are devices or bridges to other buses. Leaf nodes represent individual devices.
- Logical relationships can be created where nonleaf nodes can represent system services or subsystems whose children are the power-managed objects managed by that subsystem.

Device drivers for power-managed devices register with the power manager by attaching to a node within the namespace or by creating a new node in the appropriate location. Once attached, the driver is responsible for responding to requests from the power manager to manage the device's power mode.

The framework allows for both statically and dynamically configured systems:

- A statically configured system contains a fixed hardware configuration, where all devices that are present are known to the power manager. In this case, the power manager can “pre-populate” the namespace with entries for each device, then start all device drivers during system initialization.
- A dynamically configured system contains devices that can be dynamically added to the system (e.g. devices on a USB bus or PCMCIA controller). In this case, drivers for these devices are

often started after the power manager has initialized the system. When the drivers start up, they create nodes in the namespace to represent the new device.

How and when a driver is started makes no difference to the power manager; as soon as a driver registers with the power manager, its device will be managed according to the system's PM policy.

Power modes

The framework defines four generic power modes that are universally applied to all power-managed objects. These primarily define the operational state of the object, and by implication, its power level:

Power mode	Description
ACTIVE	A fully operational state where hardware is powered up. This is the normal operating mode.
IDLE	A partially operational state, where hardware can be partially powered. From a user's point of view, the object is fully operational; if necessary, it will become ACTIVE to service user requests that can't be handled within the IDLE power mode. This mode is typically used only for dynamic power optimizations by a driver itself. For example, if it determines that its device isn't in use, it may enter an IDLE mode to conserve power. When the device is next used, it will move back to ACTIVE.

continued...

Power mode	Description
STANDBY	A nonoperational state, where hardware is powered down. This mode is used by the power manager when the system as a whole is being powered down to a low-power standby state. Entering a STANDBY state typically requires a driver to save the context needed to reinitialize a device to an ACTIVE state (e.g. because the device hardware context or the system standby mode disables system RAM, requiring the driver to preserve the software state).
OFF	A nonoperational state, where hardware is powered off. Entering the OFF state doesn't require the driver to save any context.

Note that all power-managed objects support the four operating modes, even if there's no specific IDLE or STANDBY mode.

If there's no IDLE mode, the object implements the IDLE mode using the same operational characteristics and hardware power level as its ACTIVE mode instead.

If there's no specific STANDBY, the object implements the STANDBY mode using the same physical power level as its OFF mode. In this case, the only difference is that entry into STANDBY may require the object to save context needed to reinitialize to an ACTIVE state.

These generic power modes provide a uniform way for the power manager (or power-aware applications) to control and reason about the power levels and functionality of a power-managed object, without requiring any device-specific knowledge.

The framework allows devices to implement an arbitrary number of additional device-specific modes that are sublevels of these four generic modes. For example, a driver may define:

- *active* modes that differ in their performance and power consumption
- *idle* modes that differ in their power consumption and the latency to return to an active mode
- *standby* modes that provide system wakeup functionality.

These device-specific modes allow finer-grained control over a device's power mode and operational characteristics where appropriate.

Properties

Each power-managed object may have arbitrary properties associated with it. Each property consists of an *identifier,value* pair:

identifier An integer value used to define the property type.

value A data structure whose size and format is defined by the identifier.

The power manager's policy code is told whenever a property value is changed. This allows the policy code to respond to changes that are relevant to evaluating the optimum system power state.

The framework allows user-defined *properties* to be associated with any object, so you can define arbitrary PM-related properties or attributes that can be manipulated by the power manager or power-aware applications. For example, you can have properties defined that describe battery power levels, vehicle bus status, or other environmental conditions relevant to the embedded system.

Power-aware applications

Power-aware applications communicate with the power manager using an API implemented by the **libpm** library. The API allows the application to manipulate power-managed objects by:

- querying an object's power-mode status

- modifying an object's power-mode status (if allowed by the policy)
- receiving notification (via a SIGEVENT) when the power-mode status changes
- querying or modifying properties associated with an object.

Power-aware applications fall into two main categories:

- *system-monitoring applications* — these monitor aspects of the system that may be relevant to the PM policy (e.g. control messages on a vehicle bus, discrete events such as battery-level indications, ignition key insertion/removal, transducer readings, etc.).

This information can be represented using power manager properties. The monitoring application can update the property value to inform the power manager's PM policy. The policy can then use this information to evaluate the appropriate system power state.

- *power-sensitive applications* — these register with the power manager for notification when the state of a power-managed object changes. The application can then change its behavior as required (e.g. by enabling or disabling features based on available devices).

Power-sensitive applications can also request to change the power mode of an object, which can be used by the power manager's PM policy to intelligently decide which devices are eligible to be powered down at any given time.

Persistent storage services

Most CPUs support a low-power mode, where the CPU and peripherals are powered down with DRAM maintained in self-refresh mode. But for embedded systems that spend most of their time in a standby state for prolonged periods of time, this is unacceptable, because the power used for self-refresh over prolonged periods will exceed the reserve capacity provided by limited battery sources.

For these systems, the typical standby state will disable DRAM, so that only the limited hardware necessary to wake the system remains

powered. When the system is woken up, a warm reboot is performed; the power manager then needs to reinitialize the system according to the power-mode state it decides the system should be in. This may require drivers or power-aware applications to save volatile configuration information in persistent storage so that this information can be recovered when they are restarted.

Services for persistent storage are provided by a *persistent storage manager* that controls one or more persistent stores. These persistent stores can make use of a number of different media:

Media	Features
Shared memory	Can preserve data across application restarts. Note that this data isn't preserved across system reboots.
Raw flash partitions	Can be used when system RAM isn't preserved. This data can then be retrieved when the system is restarted.
Flash filesystem	As with a raw flash partitions, this data is preserved across system reboots.
Specialized memory ^a	Can be maintained for limited periods of time even if system RAM is disabled. Although these do require some power to maintain, the latency to save and restore persistent data is much lower than flash. This may be useful where the system must be powered down very quickly and will be powered back up within a reasonably short period of time (e.g. during an engine crank).
Custom storage	Can be implemented using an internal storage interface in the persistent storage manager.

^a For example, SRAM or a limited set of SDRAM banks that remain in self-refresh mode.

The persistent storage manager can manage multiple storage media, with one of those selected as the currently “active store.” The underlying storage is hidden from clients, who simply request to save or restore their data.

Your power manager controls the choice of media used. Given your custom set of available stores for your embedded system, your power manager policy sets the “active store” based on the nature of the standby state it is entering before powering down the system.

CPU power management

Many CPUs offer a variety of available operating modes that provide different levels of power consumption:

- Frequency or voltage scaling can reduce the power consumption of the CPU while maintaining normal instruction execution.

The system as a whole is fully operational; these different modes simply provide a tradeoff between performance and power consumption.

- The CPU can be halted, with RAM and peripherals remaining powered. The CPU will be powered up when an interrupt occurs; the latency to return to normal operation is very short, typically only a few CPU cycles.

The system as a whole is fully operational; this mode simply reduces the power consumption due to CPU activity. However, the device will still draw power for peripherals and RAM refresh.

This mode is generally used by the kernel’s idle loop to reduce power consumption when there are no runnable threads, without impacting the system’s realtime performance in responding to peripheral interrupts.

- The CPU can be placed in a low-power sleep or suspend mode where RAM is placed into self-refresh mode. The CPU will be

powered up by some configured wakeup source (e.g. an RTC, peripheral interrupt, or GPIO line).

For some system-on-chip processors, this mode may also power down on-chip peripherals, further reducing the overall power consumption.

The latency to return to normal operation is relatively long because the memory controller and on-chip peripherals may need to be reinitialized. However, the system state remains preserved in RAM — this can be a way to significantly reduce system power consumption while still providing a relatively low-latency to return to a fully active system.

- The CPU can be placed in a sleep or suspend mode where RAM is powered down, along with all on-chip peripherals. The CPU will be powered up by an appropriately configured wakeup source (e.g. an RTC or GPIO line).

Returning to normal operation is essentially a complete system reboot, because all system state information preserved in RAM is lost. However, this provides the lowest power consumption of all, requiring power for only those components required to wake up the CPU.

The power manager's PM policy determines which CPU modes are appropriate for the various system states required for your device. Remember that this policy is user-written — as the embedded system designer, you are in the best position to decide on the optimum balance between power consumption and performance or latency to return to an operational mode.

These CPU modes are highly processor-specific. Also, the nature of the wakeup sources depends on the system hardware implementation. For this reason, the implementation is provided by the *power()* callout within the BSP-specific startup program, and may also require support from the BSP-specific IPL (if the CPU wakeup causes a warm reboot via the reset vector).

Reference implementations provide the basic CPU-level support required for the modes supported by a particular CPU. You may then

customize this as required to handle board-specific wakeup configuration, if necessary.

Chapter 15

The Photon microGUI

In this chapter...

A graphical microkernel	311
The Photon event space	312
Graphics drivers	317
Font support	319
Unicode multilingual support	320
Animation support	321
Multimedia support	322
Printing support	324
The Photon Window Manager	324
Widget library	325
Driver development kits	342
Summary	342



A graphical microkernel



This chapter provides an overview of the Photon microGUI, the graphical environment for QNX Neutrino. For more information, see the Photon documentation set.

Many embedded systems require a UI so that users can access and control the embedded application. For complex applications, or for maximum ease of use, a graphical windowing system is a natural choice. However, the windowing systems on desktop PCs simply require too much in the way of system resources to be practical in an embedded system where memory and cost are limiting factors.

Drawing upon the successful approach of the QNX Neutrino microkernel architecture to achieve a POSIX OS environment for embedded systems, we have followed a similar course in creating the Photon microGUI windowing system.

To implement an effective microkernel OS, we first had to tune the microkernel so that the kernel calls for IPC were as lean and efficient as possible (since the performance of the whole OS rests on this message-based IPC). Using this low-overhead IPC, we were able to structure a GUI as a graphical “microkernel” process with a team of cooperating processes around it, communicating via that IPC.

While at first glance this might seem similar to building a graphical system around the classic client/server paradigm already used by the X Window System, the Photon architecture differentiates itself by restricting the functionality implemented within the graphical microkernel (or server) itself and distributing the bulk of the GUI functionality among the cooperating processes.

The Photon microkernel runs as a tiny process, implementing only a few fundamental primitives that external, optional processes use to construct the higher-level functionality of a windowing system. Ironically, for the Photon microkernel itself, “windows” do not exist. Nor can the Photon microkernel “draw” anything, or manage a pen, mouse, or keyboard.

To manage a GUI environment, Photon creates a 3-dimensional *event space* and confines itself only to handling *regions* and processing the clipping and steering of various events as they flow through the regions in this event space.

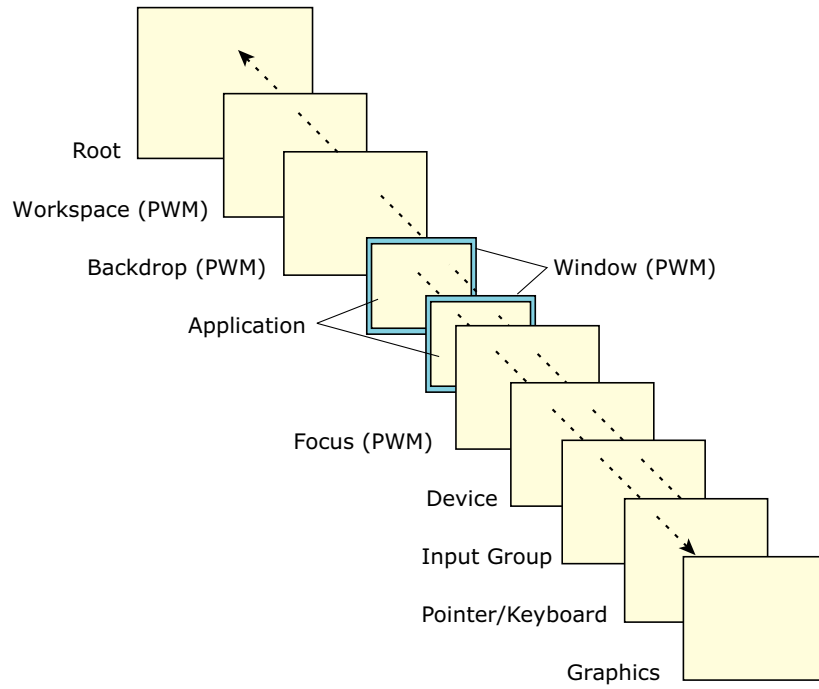
This abstraction is roughly parallel to the concept of a microkernel OS being incapable of filesystem or device I/O, but relying instead on external processes to provide these high-level services. Just as this allows a microkernel OS to scale up or down in size and functionality, so also a microkernel GUI.

The core microkernel “abstraction” implemented by the Photon microkernel is that of a graphical event space that other processes can populate with regions. Using native IPC to communicate with the Photon microkernel, these other processes manipulate their regions to provide higher-level graphical services or to act as user applications. By *removing* service-providing processes, Photon can be scaled down for limited-resource systems; by *adding* service-providing processes, Photon can be scaled up to full desktop functionality.

The Photon event space

The “event space” managed by the Photon microkernel can be visualized as an empty void with a “root region” at the back. The end-user can be imagined to be “looking into” this event space from the front. Applications place “regions” into the 3-dimensional space between the root region and the end-user; they use those regions to generate and accept various types of “events” within this space.

Processes that provide device driver services place regions at the front of the event space. In addition to managing the event space and root region, the Photon microkernel projects *draw events*.



Photon regions.

We can think of these events that travel through this space as “photons” (from which this windowing system gets its name). Events themselves consist of a list of rectangles with some attached data. As these events flow through the event space, their rectangle lists intersect the “regions” placed there by various processes (applications).

Events traveling away from the root region of the event space are said to be traveling outwards (or towards the user), while events from the user are said to be traveling inwards towards the root region at the back of the event space.

The interaction between events and regions is the basis for the input and output facilities in Photon. Pen, mouse, and keyboard events travel away from the user towards the root plane. Draw events

originate from regions and travel towards the device plane and the user.

Regions

Regions are managed in a hierarchy associated as a family of rectangles that define their location in the 3-dimensional event space. A region also has attributes that define how it interacts with various classes of events as they intersect the region. The interactions a region can have with events are defined by two bitmasks:

- *sensitivity* bitmask
- *opaque* bitmask.

The sensitivity bitmask uses specific event types to define which intersections the process owning the region wishes to be informed of. A bit in the sensitivity bitmask defines whether or not the region is sensitive to each event type. When an event intersects a region for which the bit is set, a copy of that event is enqueued to the application process that owns the region, notifying the application of events traveling through the region. This notification doesn't modify the event in any way.

The opaque bitmask is used to define which events the region can or can't pass through. For each event type, a bit in the opaque mask defines whether or not the region is opaque or transparent to that event. The optical property of "opaqueness" is implemented by modifying the event itself as it passes through the intersection.

These two bitmasks can be combined to accomplish a variety of effects in the event space. The four possible combinations are:

Bitmask combination:	Description:
Not sensitive, transparent	The event passes through the region, unmodified, without the region owner being notified. The process owning the region simply isn't interested in the event.

continued...

Bitmask combination:	Description:
Not sensitive, opaque	The event is clipped by the region as it passes through; the region owner isn't notified. For example, most applications would use this attribute combination for draw event clipping, so that an application's window wouldn't be overwritten by draw events coming from underlying windows.
Sensitive, transparent	A copy of the event is sent to the region owner; the event then continues, unmodified, through the event space. A process wishing to log the flow of events through the event space could use this combination.
Sensitive, opaque	A copy of the event is sent to the region owner; the event is also clipped by the region as it passes through. By setting this bitmask combination, an application can act as an event filter or translator. For every event received, the application can process and regenerate it, arbitrarily transformed in some manner, possibly traveling in a new direction, and perhaps sourced from a new coordinate in the event space. For example, a region could absorb pen events, perform handwriting recognition on those events, and then generate the equivalent keystroke events.

Events

Like regions, events also come in various classes and have various attributes. An event is defined by:

- an originating region
- a type
- a direction
- an attached list of rectangles
- some event-specific data (optional).

Unlike most windowing systems, Photon classifies *both* input (pen, mouse, keyboard, etc.) and output (drawing requests) as events. Events can be generated either from the regions that processes have placed in the event space or by the Photon microkernel itself. Event types are defined for:

- keystrokes
- pen and mouse button actions
- pen and mouse motion
- region boundary crossings
- expose and covered events
- draw events
- drag events
- drag-and-drop events.

Application processes can either poll for these events, block and wait for them to occur, or be asynchronously notified of a pending event.

The rectangle list attached to the event can describe one or more rectangular regions, or it can be a “point-source” — a single rectangle where the upper-left corner is the same as the lower-right corner.

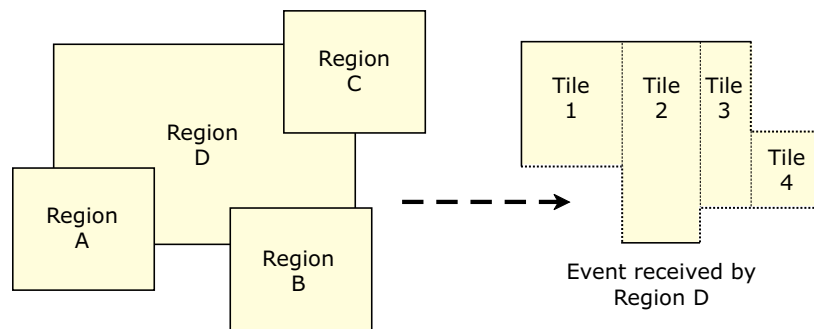
When an event intersects a region that is opaque to it, that region’s rectangle is “clipped out” of the event’s list of rectangles such that the list describes only the portion of the event that would ultimately be visible.

The best way to illustrate how this clipping is performed is to examine the changes in the rectangle list of a draw event as it passes through various intersecting regions. When the draw event is first generated, the rectangle list consists of only a single, simple rectangle describing the region that the event originated from.

If the event goes through a region that clips the upper-left corner out of the draw event, the rectangle list is modified to contain only the

two rectangles that would define the area remaining to be drawn. These resulting rectangles are called “tiles.”

Likewise, every time the draw event intersects a region opaque to draw events, the rectangle list will be modified to represent what will remain visible of the draw event after the opaque region has been “clipped out.” Ultimately, when the draw event arrives at a graphics driver ready to be drawn, the rectangle list will precisely define only the portion of the draw event that is to be visible.



Opaque regions are clipped out.

If the event is entirely clipped by the intersection of a region, the draw event will cease to exist. This mechanism of “opaque” windows modifying the rectangle list of a draw event is how draw events from an underlying region (and its attached process) are properly clipped for display as they travel towards the user.

Graphics drivers

Graphics drivers are implemented as processes that place a region at the front of the event space. Rather than inject pen, mouse, or keyboard events (as would an input driver), a graphics driver’s region is *sensitive* to draw events coming out of the event space. As draw events intersect the region, those events are received by the graphics driver process. In effect, the region can be imagined to be coated in “phosphor,” which is illuminated by the impact of “photons.”

Since the Photon drawing API accumulates draw requests into batches emitted as single draw events, each draw event received by the driver contains a list of individual graphical primitives to be rendered. By the time the draw event intersects the graphics driver region, its rectangle list will also contain a “clip list” describing exactly which portions of the draw list are to be rendered to the display. The driver’s job is to transform this clipped draw list into a visual representation on whatever graphics hardware the driver is controlling.

One advantage of delivering a “clip list” within the event passed to the driver is that each draw request then represents a significant “batch” of work. As graphics hardware advances, more and more of this “batch” of work can be pushed directly into the graphics hardware. Many display controller chips already handle a single clip rectangle; some handle multiple clip rectangles.

Although using the OS IPC services to pass draw requests from the application to the graphics driver may appear to be an unacceptable overhead, our performance measurements demonstrate that this implementation performs as well as having the application make direct calls into a graphics driver. One reason for such performance is that multiple draw calls are batched with the event mechanism, allowing the already minimal overhead of our lightweight IPC to be amortized over many draw calls.

Multiple graphics drivers

Since graphics drivers simply put a region into the Photon event space, and since that region describes a rectangle to be intersected by draw events, it naturally follows that multiple graphics drivers can be started for multiple graphics controller cards, all with their draw-event-sensitive regions present in the same event space.

These multiple regions could be placed adjacent to each other, describing an array of “drawable” tiles, or overlapped in various ways. Since Photon inherits the OS’s network transparency, Photon applications or drivers can run on any network node, allowing additional graphics drivers to extend the graphical space of Photon to the physical displays of many networked computers. By having the

graphics driver regions overlap, the draw events can be replicated onto multiple display screens.

Many interesting applications become possible with this capability. For example, a factory operator with a wireless-LAN handheld computer could walk up to a workstation and drag a window from a plant control screen onto the handheld, and then walk out onto the plant floor and interact with the control system.

In other environments, an embedded system without a UI could project a display onto any network-connected computer. This connectivity also enables useful collaborative modes of work for people using computers — a group of people could simultaneously see the same application screen and cooperatively operate the application.

From the application's perspective, this looks like a single unified graphical space. From the user's perspective, this looks like a seamlessly connected set of computers, where windows can be dragged from physical screen to physical screen across network links.

Color model

Colors processed by the graphics drivers are defined by a 24-bit RGB quantity (8 bits for each of red, green, and blue), providing a total range of 16,777,216 colors. Depending on the actual display hardware, the driver will either invoke the 24-bit color directly from the underlying hardware or map it into the color space supported by the less-capable hardware.

Since the graphics drivers use a hardware-independent color representation, applications can be displayed without modifications on hardware that has varied color models. This allows applications to be “dragged” from screen to screen, without concern for what the underlying display hardware's color model might be.

Font support

Photon uses Bitstream's Font Fusion object-oriented font engine, which provides developers with full font fidelity and high-quality

typographic output at any resolution on any device, while maintaining the integrity of the original character shapes.

Photon is shipped with a limited number of TrueType fonts. These industry-standard fonts are readily available from various sources.

Stroke-based fonts

To support Asian languages (e.g. Chinese, Japanese, and Korean), Photon relies on Bitstream's stroke-based fonts. These high-speed fonts are ideal for memory-constrained environments. For example, a complete traditional Chinese font with over 13,000 characters can occupy as much as 8M in a conventional desktop system — a stroke-based version of the same font occupies less than 0.5M!

Apart from their compact size and fast rasterization, these fonts are also fully *scalable*, which makes them perfect for various nondesktop displays such as LCDs, TVs, PDAs, and so on.

Unicode multilingual support

Photon is designed to handle international characters. Following the Unicode Standard (ISO/IEC 10646), Photon provides developers with the ability to create applications that can easily support the world's major languages and scripts.



Scripts that read from right to left, such as Arabic, aren't supported at this time.

Unicode is modeled on the ASCII character set, but uses a 16-bit (or 32-bit) encoding to support full multilingual text. There's no need for escape sequences or control codes when specifying any character in any language. Note that Unicode encoding conveniently treats all characters — whether alphabetic, ideographs, or symbols — in exactly the same way.

UTF-8 encoding

Formerly known as UTF-2, the UTF-8 (for “8-bit form”) transformation format is designed to address the use of Unicode character data in 8-bit UNIX environments.

Here are some of the main features of UTF-8:

- Unicode characters from U+0000 to U+007E (ASCII set) map to UTF-8 bytes 00 to 7E (ASCII values).
- ASCII values don’t otherwise occur in a UTF-8 transformation, giving complete compatibility with historical filesystems that parse for ASCII bytes.
- The first byte indicates the number of bytes to follow in a multibyte sequence, allowing for efficient forward parsing.
- Finding the start of a character from an arbitrary location in a byte stream is efficient, because you need to search at most four bytes backwards to find an easily recognizable initial byte. For example:
`isInitialByte = ((byte & 0xC0) != 0x80);`
- UTF-8 is reasonably compact in terms of the number of bytes used for encoding.
- UTF-8 strings are terminated with a single NULL byte, like traditional ASCII C strings.

Animation support

Photon provides flicker-free animations by employing off-screen video memory where possible. For instance, a special container widget (**PtOSContainer**) creates a dedicated off-screen memory context for drawing images. The **PtOSContainer** widget uses a block of video memory large enough to hold an image the size of its canvas. (For more information about widgets, see the section “Widget library” in this chapter.)

Photon’s graphics drivers also maximize the use of off-screen memory to enhance the perceptual performance of animated images. The

graphics drivers support other advanced techniques, such as direct graphics mode, alpha-blending, chroma-key substitution, and more.

Video overlay

Besides the ability to superimpose a semi-transparent image on top of a background (alpha-blending) or to place a color-masked foreground image on top of a separate background (chroma-key), Photon also supports *video overlay* — a full-motion video image is rendered within a window on the display.

Layers

Some display controllers let you transparently overlay multiple “screens” on a single display. Each overlay is called a *layer*.

You can use layers to combine independent display elements. Since the graphics hardware performs the overlaying, this can be more efficient than rendering all of the display elements onto a single layer. For example, a fast navigational display can have a scrolling navigational map on a background layer and a web browser or other popup GUI element on a foreground layer.

The images on all the active layers of a display are combined, using alpha-blending, chroma-keying, or both, to produce the final image on the display.

Multimedia support

Many developers of Photon applications need to target resource-constrained embedded systems. While conventional multimedia solutions rely on powerful CPUs and graphics hardware, Photon’s multimedia support is ideal for embedded systems equipped with “low-end” processors (e.g. Cyrix MediaGX 200) and limited system RAM (e.g. 16M).

Plugin architecture

Photon provides a suite of multimedia plugins that are as fast and light as possible, yet able to deliver the functionality of high-end systems.

Using Photon's extensible plugin architecture and media API, developers can easily integrate their own multimedia components.

To enable embedded systems to play audio/video files without depending on disk-based access, Photon supports *streaming* technologies (MPEG-1 System Stream and MPEG-2 Program Stream).

Media Player

This Photon application forms the core of our multimedia support. Its interface is simple — the user selects an audio or video file to play, and the Media Player “plays” it. But behind the scenes, the Media Player:

- identifies and loads/unloads the appropriate plugins
- initializes the plugins
- translates user input into plugin commands
- handles plugin-generated events (callbacks)
- identifies minor file types (registry bypass)
- supports playlists
- provides the GUI for QNX plugins
- provides OEM panes for custom plugins.

Media Player plugins

Currently, the multimedia plugins include:

- Audio CD Player — a multitrack plugin that plays audio CDs
- MPEG Audio Player — decodes MPEG-1 audio layers 1, 2, and 3 (MP3); also supports MPEG-2 low-bit rate streams and MPEG 2.5 audio.
- MPEG Audio/Video Player — decodes MPEG-1 System and MPEG-2 Program streams

- MPEG Video Player — decodes MPEG-1 video streams
- Soundfile Player — plays simple audio files; supports Wave, AIFF, and other formats.



QNX Software Systems provides licenses for some, but not all, of the above technologies. For details, contact your sales representative.

Printing support

Photon provides built-in printing support for a variety of outputs, including:

- bitmap files
- PostScript
- Hewlett-Packard PCL
- Epson ESC/P2
- Epson IJS
- Canon
- Lexmark

Photon also comes with a print-selection widget/convenience dialog to make printing simpler within developers' own applications.

The Photon Window Manager

Adding a window manager to Photon creates a fully functional desktop-class GUI. The window manager is entirely optional and can be omitted for most classes of embedded systems. If present, the window manager allows the user to manipulate application windows by resizing, moving, and iconifying them.

The window manager is built on the concept of filtering events with additional regions placed behind application regions, upon which a

title bar, resize handles, and other elements are drawn and interacted with. Since the replaceable window manager implements the actual “look and feel” of the environment, various UI flavors can be optionally installed.

Widget library

Photon provides a library of components known as *widgets* — objects that can manage much of their on-screen behavior automatically, without explicit programming effort. As a result, a complete application can be quickly assembled by combining widgets in various ways and then attaching C code to the appropriate callbacks the widgets provide.


The Photon Application Builder (PhAB), which is included as part of the Photon development system, provides an extensive widget palette in its visual development environment.

Photon provides a wide range of widgets, which can be classified as follows:

- fundamental widgets (e.g. a button)
- container widgets (e.g. a window widget)
- advanced widgets (e.g. an HTML display widget).

Fundamental widgets

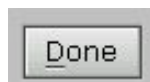
Label widget (`PtLabel`)



Active Link Color

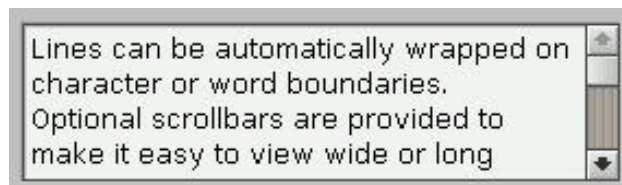
The label widget can display bitmaps, images, or textual information. The `PtLabel` widget is the superclass for all text-based widgets, providing many customizable attributes (e.g. font typeface, pop-up balloons, colors, borders, alignment, margins, etc.), all of which are inherited by all its subclasses.

Push-button widget (**PtButton**)



Push buttons are a necessary component in every windowing system. They have a raised look that changes to depressed when pushed, giving a visual cue to let the user know the button has been selected. In addition to this visual behavior, push buttons automatically invoke an application-defined callback when they're selected.

Text input widgets (**PtText**, **PtMultiText**)



Photon provides two text-input widgets:

- a simple single-line input widget (**PtText**) commonly used in forms
- a powerful wordprocessor-like multi-line widget (**PtMultiText**) providing full editing capabilities, word wrapping, automatic scrolling, and multi-font line segments.

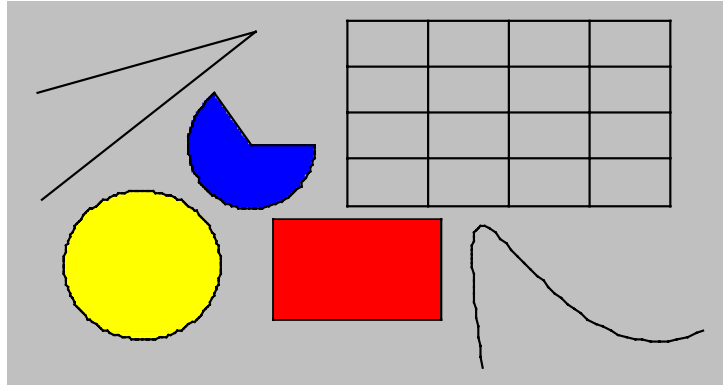
Toggle-button widgets (**PtToggleButton**)



Toggle buttons are objects that display two states — on or off. Photon provides various styles of toggle buttons, each with a different visual

appearance. Toggle buttons are used to display or request state information related to a command or action about to be performed.

Graphical widgets (**PtArc**, **PtPixel**, **PtRect**, **PtLine**, **PtPolygon**, **PtEllipse**, **PtBezier**, **PtGrid**)



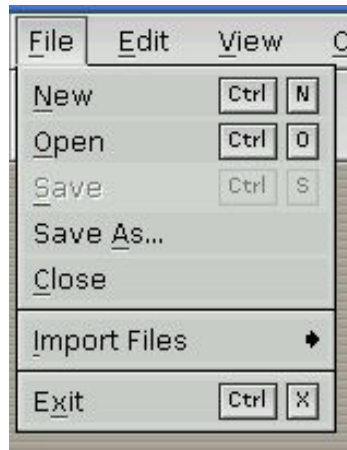
Photon has no shortage of graphical widgets. There's a widget to accomplish everything from simple lines and rectangles to complex multi-segmented Bézier curves. Graphical widgets provide attributes for color, fills, patterns, line thickness, joins, and much more.

Scrollbar widget (**PtScrollbar**)



A scrollbar widget is used to scroll the display of a viewable area. The scrollbar is combined with other widgets (e.g. **PtList**, **PtScrollContainer**) to allow scrolling.

Separator widget (**PtSeparator**)



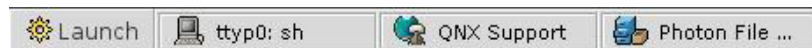
The separator widget is used to separate two or more different areas, such as the menu items as shown in this example. The separator can be customized for many different styles and looks.

Slider widget (**PtSlider**)



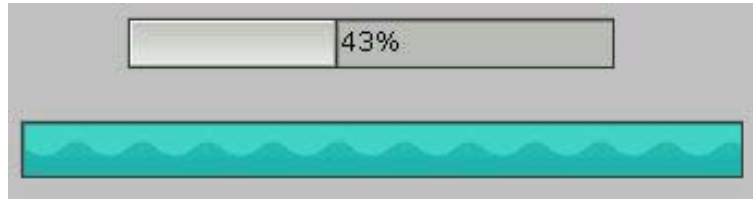
Sliders are different from scrollbars. A scrollbar defines a range, whereas a slider defines *a single value*.

Image widgets (**PtLabel**, **PtButton**)



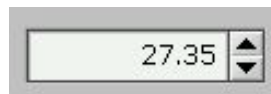
Photon supports every major graphics file standard, so you can import graphics and display them inside widgets. Many Photon widgets directly support displaying graphics — the most common are **PtButton** for making push-button toolbars and **PtLabel** for displaying images.

Progress-bar widget (**PtProgress**)



If an application needs to do something that takes a fair amount of time (e.g. loading a file), it can use the progress bar widget to let the user know what's happening and, more importantly, how much longer the process is going to take. The progress bar has many attributes for customization — it can be horizontal or vertical, it can display specific or indeterminate values (both are shown here), etc.

Numeric widgets (**PtNumericInteger**, **PtNumericFloat**)

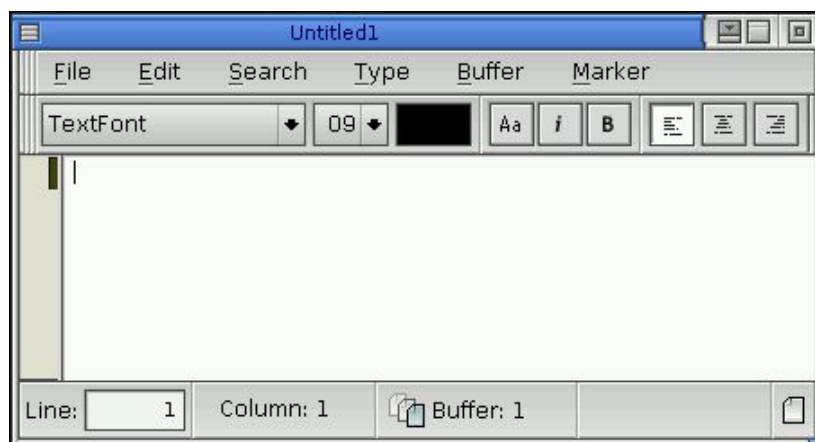


The **PtNumericInteger** class lets the user specify integer values between given minimum and maximum values. The **PtNumericFloat** class lets the user enter floating-point values.

Container widgets

A container widget is a powerful and convenient interface tool that holds other widgets and manages their layout. Containers are used extensively in most Photon applications.

Window widget (PtWindow)



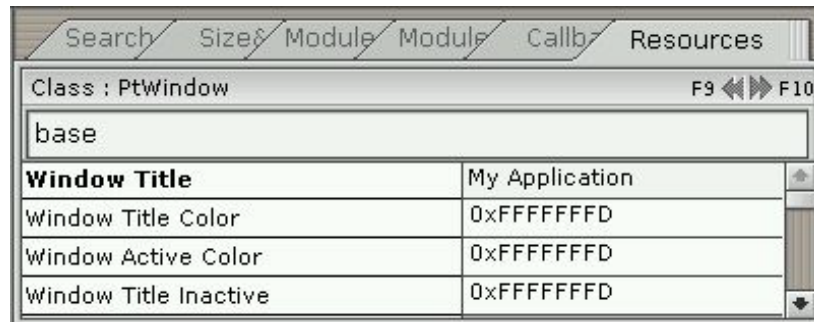
Windows are the main application containers. The main UI components (menu bars, toolbars, etc.) appear with the window widget. The widget automatically handles all the necessary interactions with the Photon Window Manager (PWM) — all you need to specify is what should and shouldn't be rendered or managed.

Group widget (PtGroup)



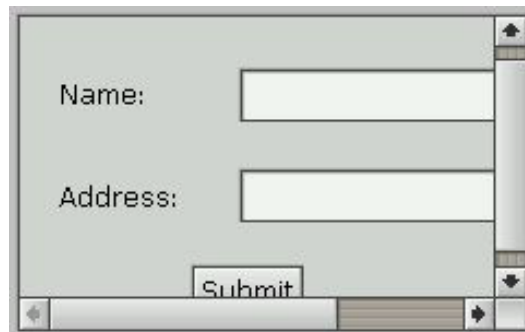
The group widget is a very powerful widget that manages the geometry of all its child widgets. It can align the widgets horizontally, vertically, or in a matrix. The widget also provides attributes that let you specify whether the children should be stretched to fit the group if it's resized larger due to anchoring.

Panel group widget (**PtPanelGroup**)



The panel group widget is a container for panels, a useful element for organizing data in dialogs. Besides managing the geometry and layout of panels, **PtPanelGroup** also provides two selection modes to switch between panels: multiple-tab selection (each panel has its own tab to select) and single-tab selection (clicking the tab pops up a menu to select other panels).

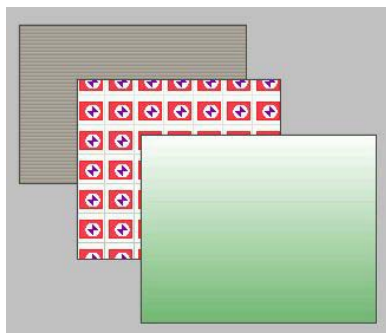
Viewport widget (**PtScrollContainer**)



The **PtScrollContainer** widget is a very powerful widget that provides a viewport into a potentially larger container. You can place any number of widgets inside a scroll container and it will automatically display a scrollbar if the widgets are contained within the viewable area. **PtScrollContainer** widgets could be used to implement a text file viewer, wordprocessor, customized list display, and so on.

To scroll child widgets quickly, the scrolling area widget uses a hardware blitter (provided the underlying graphics driver supports it).

Background widget (PtBkgd)

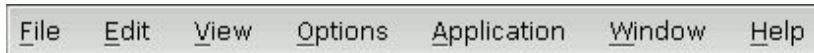


The background widget provides a way to create fancy background displays, from simple color gradations to tiled textures. This widget can handle just about any background requirement.

Advanced widgets

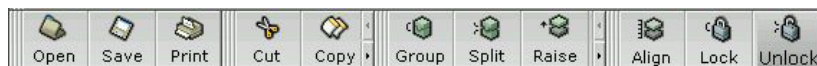
Armed with Photon's rich set of widgets, developers can build practically any graphical application imaginable. Here are some of the more powerful widgets at your disposal:

Menu-related widgets (PtMenu, PtMenuBar, PtMenuButton)



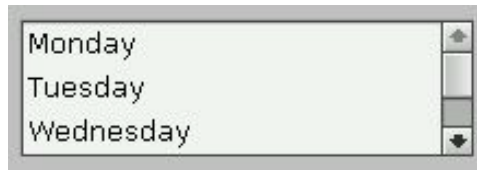
Photon has a widget for every menu-related requirement. There's a widget to simplify the creation of a standard menu bar. The menu widget handles the pop-up display, press-drag-release, point and click, keyboard traversal, and selection of menu items. The menu button widget is used for creating individual menu items.

Toolbar widgets (PtToolbar, PtToolbarGroup)



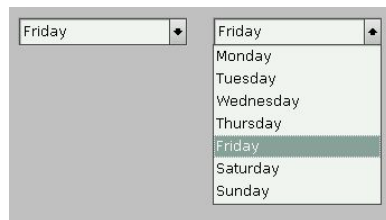
This container holds buttons, labels, images, whatever widgets you wish, and aligns them either vertically or horizontally in a toolbar. The toolbar group widget lets you combine a toolbar with a menu bar to create a very flexible access element for your applications.

List widget (**PtList**)



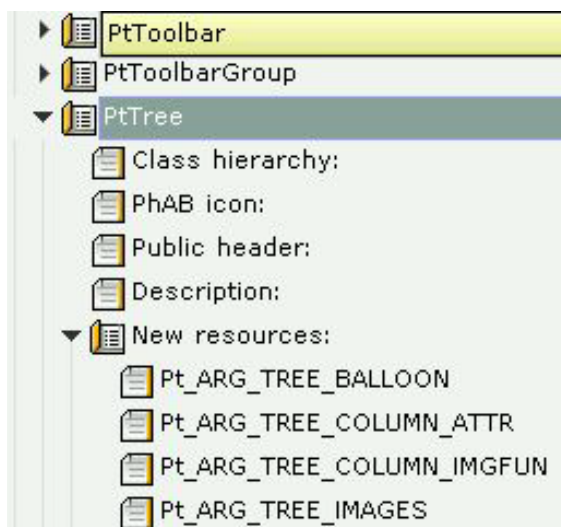
The list widget is a very powerful widget that manages a list of items. It provides many different selection modes, including single selection, multiple selection and range selection. The list widget also supports multi-columnned lists through the use of a divider widget (**PtDivider**).

Pulldown list widget (**PtComboBox**)



The pulldown list widget combines the **PtText** widget (for text input) with a pulldown button for displaying a list widget. When the user selects from the list, the text widget is automatically updated with the current selection. The pulldown list widget is very useful for displaying a list of items using a small space. Dialogs and containers use a lot less screen real-estate, which is important in embedded environments.

Tree widget (PtTree)



The tree widget is similar to the list widget — in fact they both have the same ancestors. The main difference is that the tree widget displays the items in a hierarchical manner. Items, called branches, can be expanded or collapsed; any number of tree branches can be created. Each branch can define its own unique image to display. Trees are useful because they display information in a very logical manner.

Photon applications that use the tree widget include the File Manager (directory display), PhAB (widget layout), Helpviewer, and many others.

Terminal widgets (PtTty, PtTerminal)

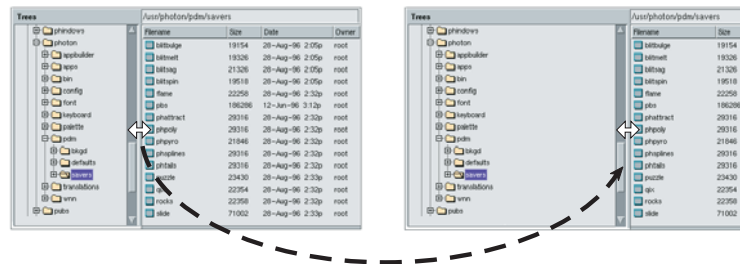
```
login
logname
logout
ls
m4
mailq
make
makeinfo
man
md5sum
melt
merge
mesg
messages
mkasmoff
# pwd
/etc/root
# _

yes
zcat
zcmp
zdiff
zforce
zgrep
zip
zipcloak
zipinfo
zipnote
zipsplit
zmore
znew
```

A terminal widget creates and manages an entire text-mode terminal inside a widget. Just drop it into your application and you've created your very own **pterm** (Photon's terminal application).

The terminal widget doesn't stop there — it also provides complete cut-and-paste functionality and quick-launch help by highlighting any text within the widget.

Divider widget (PtDivider)

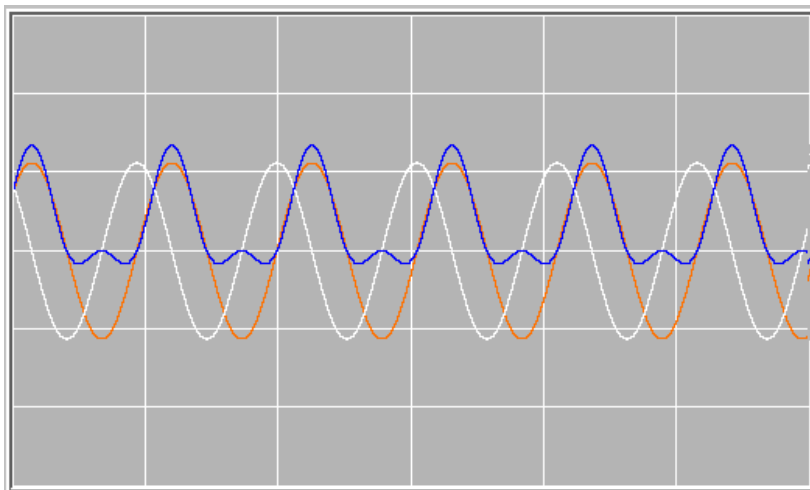


This powerful widget manages its children in a unique and useful way. When you place two or more widgets inside a divider widget, it automatically puts little separators in between the child widgets. Using these separators, the user can drag back and forth, causing the

child widgets on either side of the separator to be resized. This is very useful for creating resizable column headings for lists. In fact, if you drop a divider widget into a list widget, it will automatically turn your simple list into a resizable multi-column list.

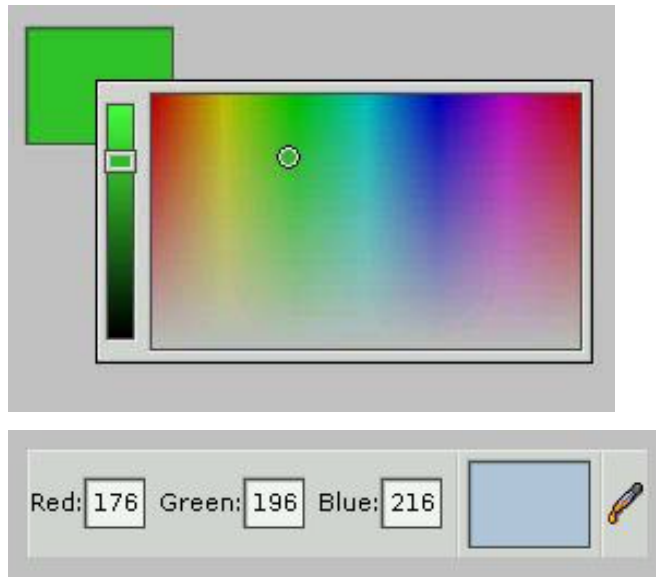
Dividers aren't limited to just labels or buttons. Any widgets can be placed inside to create side-by-side resizable trees, scroll areas, and so on.

Trend graph widgets (**PtTrend** and **PtMTrend**)



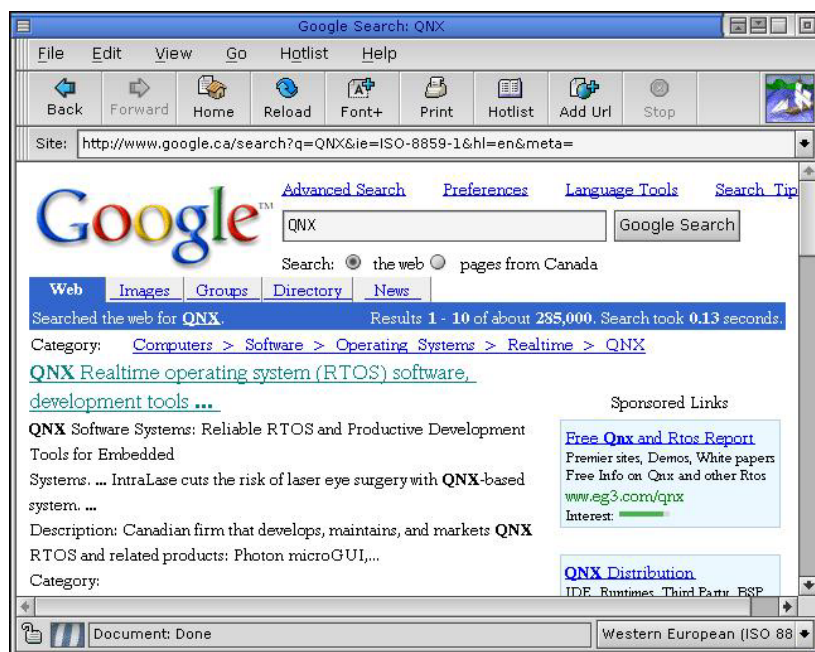
Realtime systems often require trend graphs. Photon comes with a trend bar widget, **PtTrend**, that supports the display of multiple trend lines simultaneously. If your graphics hardware supports masked blits, it can even smooth-scroll the trend across grid lines. The **PtMTrend** widget has additional features, such as a trace line, which make it appropriate for medical applications.

Color-selection widgets (PtColorSel, PtColorPanel, PtColorPatch, PtColorSelGroup, PtColorWell)



Photon provides several handy controls for building color-selection dialogs. This convenient set of widgets includes **PtColorPanel**, a compound widget that provides several ways to easily select a color.

Web client widget (PtWebClient)



The **PtWebClient** widget is used to start, interact, and control a web browser. The widget also provides a user-defined area within the application for the server to format and display web pages.

The application controls the server by setting widget resources. The server communicates status information and user interaction back to the application using the widget callbacks. The **PtWebClient** widget transparently supports the version of HTML that the server supports.

Convenience functions

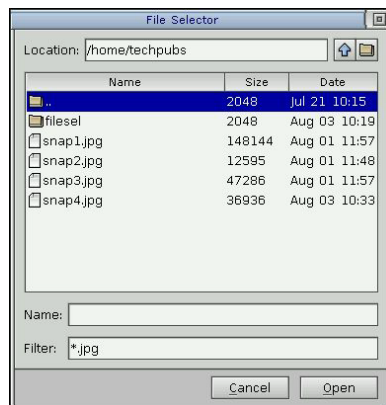
Once a widget has been created, you can take advantage of Photon's convenience functions to easily set up dialogs and control the widget.

Here are some examples of common dialogs created using the following convenience functions in Photon's widget toolkit:

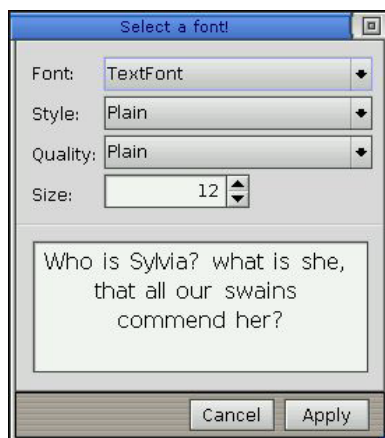
- *PtFileSelection()* — create a file-selector dialog

- *PtFontSelection()* — display a modal dialog for selecting a font
- *PtPrintSelection()* — display a modal dialog for selecting print options
- *PtAlert()* — display a message and request a response from the user
- *PtNotice()* — display a message and wait for acknowledgment by the user
- *PtPrompt()* — display a message and get textual input from the user.

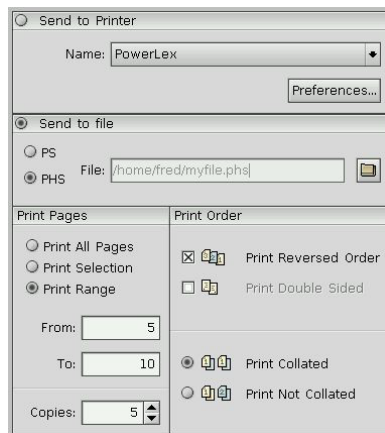
File-selection dialog (*PtFileSelection()*)



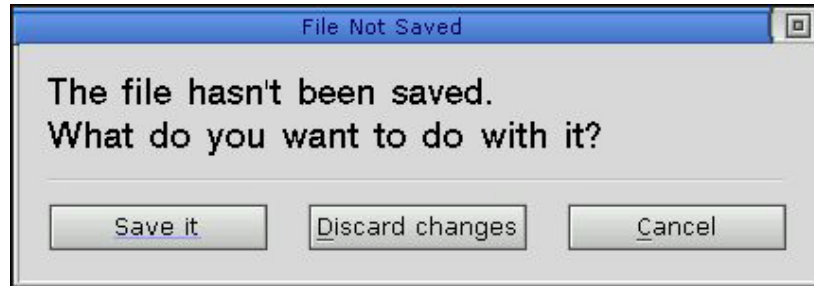
The *PtFileSelection()* function incorporates a tree widget that displays files, directories, links to files or directories, and custom entries. Besides selecting a particular file in response to an application prompt, users can also navigate an entire filesystem and choose their own file and directory.

Font-selection dialog (*PtFontSelection()*)

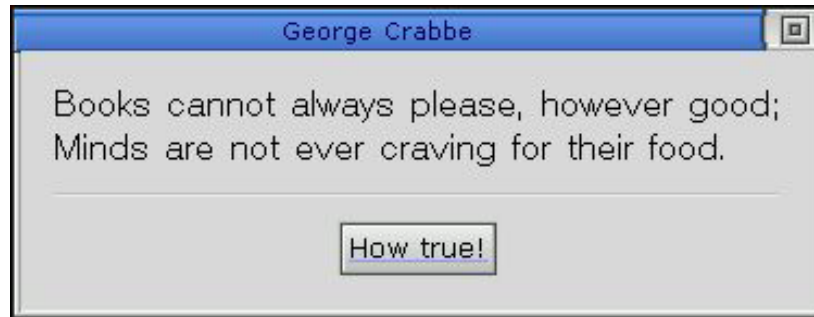
To accommodate the wide selection of fonts available, Photon provides a handy font-selector. The dialog displays a list of all available fonts and allows the user to choose the typeface and style (bold, italic, etc.) and to indicate whether the font should be anti-aliased.

Print-selection dialog (*PtPrintSelection()*)

The print selector lets a user select a printer or control its properties. The user may also select a range of pages as well as the number of copies to print.

Alert dialog (*PtAlert()*)

This modal dialog is useful for informing the user of some significant event and waiting for a response (e.g. clicking a button).

Notice dialog (*PtNotice()*)

This dialog can display a message that may or may not require a response. This type of dialog often contains an "OK" button for the user to acknowledge the notice.

Prompt dialog (*PtPrompt()*)

Like the alert dialog, the prompt dialog displays a message that requires the user to respond, but it also provides a field for inputting text within the dialog.

Driver development kits

As a product geared towards developers, Photon offers all the tools needed to build high-performance, accelerated graphics drivers that can be readily tailored for particular graphics cards and chipsets.

Developers will be able to create drivers that support advanced graphics techniques (e.g. alpha-blending or chroma-key substitution) through a software-only approach, a perfect fallback for “simple” hardware that doesn’t directly handle such techniques.

The Photon graphics driver development kit provides full source code for several sample drivers as well as detailed instructions for handling the hardware-dependent issues involved in developing custom drivers.

Summary

Photon represents a new approach to GUI building — using a microkernel and a team of cooperating processes, rather than the monolithic approach typified by other windowing systems. As a result, Photon exhibits a unique set of capabilities:

- Low memory requirements enable Photon to deliver a high level of windowing functionality to environments where only a graphics library might have been allowed within the memory constraints.
- Photon provides a very flexible, user-extensible architecture that allows developers to extend the GUI in directions unique to their applications.
- With flexible cross-platform connectivity, Photon applications can be used from virtually any connected desktop environment.



Glossary



A20 gate

On x86-based systems, a hardware component that forces the A20 address line on the bus to zero, regardless of the actual setting of the A20 address line on the processor. This component is in place to support legacy systems, but the QNX Neutrino OS doesn't require any such hardware. Note that some processors, such as the 386EX, have the A20 gate hardware built right into the processor itself — our IPL will disable the A20 gate as soon as possible after startup.

adaptive

Scheduling algorithm whereby a thread's priority is decayed by 1. See also **FIFO**, **round robin**, and **sporadic**.

atomic

Of or relating to atoms. :-)

In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered *indivisible*. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread's position.

attributes structure

Structure containing information used on a per-resource basis (as opposed to the **OCB**, which is used on a per-open basis).

This structure is also known as a **handle**. The structure definition is fixed (**iofunc_attr_t**), but may be extended. See also **mount structure**.

bank-switched

A term indicating that a certain memory component (usually the device holding an **image**) isn't entirely addressable by the processor. In this case, a hardware component manifests a small portion (or "window") of the device onto the processor's address bus. Special

commands have to be issued to the hardware to move the window to different locations in the device. See also **linearly mapped**.

base layer calls

Convenient set of library calls for writing resource managers. These calls all start with *resmgr_**(*).* Note that while some base layer calls are unavoidable (e.g. *resmgr_pathname_attach()*), we recommend that you use the **POSIX layer calls** where possible.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an “extension” to the BIOS or ROM Monitor — control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes 0x55 and 0xAA must be present (in that order) as the first two bytes in the device, with control being transferred to offset 0x0003.

block-integral

The requirement that data be transferred such that individual structure components are transferred in their entirety — no partial structure component transfers are allowed.

In a resource manager, directory data must be returned to a client as **block-integral** data. This means that only complete **struct dirent** structures can be returned — it’s inappropriate to return partial structures, assuming that the next `_IO_READ` request will “pick up” where the previous one left off.

bootable

An image can be either bootable or **nonbootable**. A bootable image is one that contains the startup code that the IPL can transfer control to.

bootfile

The part of an OS image that runs the **startup code** and the Neutrino microkernel.

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()*).

CIFS

Common Internet File System (aka 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.

combine message

A resource manager message that consists of two or more messages. The messages are constructed as combine messages by the client's C library (e.g. *stat()*, *readblock()*), and then handled as individual messages by the resource manager.

The purpose of combine messages is to conserve network bandwidth and/or to provide support for atomic operations. See also **connect message** and **I/O message**.

connect message

In a resource manager, a message issued by the client to perform an operation based on a pathname (e.g. an **io_open** message). Depending on the type of connect message sent, a context block (see **OCB**) may be associated with the request and will be passed to subsequent I/O messages. See also **combine message** and **I/O message**.

connection

A kernel object used with message passing.

Connections are created by client threads to “connect” to the channels made available by servers. Once connections are established, clients can *MsgSendv()* messages over them. If a number of threads in a process all attach to the same channel, then the one connection is shared among all the threads. Channels and connections are identified within a process by a small integer.

The key thing to note is that connections and file descriptors (**FD**) are one and the same object. See also **channel** and **FD**.

context

Information retained between invocations of functionality.

When using a resource manager, the client sets up an association or **context** within the resource manager by issuing an *open()* call and getting back a file descriptor. The resource manager is responsible for storing the information required by the context (see **OCB**). When the client issues further file-descriptor based messages, the resource manager uses the OCB to determine the context for interpretation of the client's messages.

cooked mode

See **canonical mode**.

core dump

A file describing the state of a process that terminated abnormally.

critical section

A code passage that *must* be executed “serially” (i.e. by only one thread at a time). The simplest form of critical section enforcement is via a **mutex**.

deadlock

A condition in which one or more threads are unable to continue due to resource contention. A common form of deadlock can occur when one thread sends a message to another, while the other thread sends a message to the first. Both threads are now waiting for each other to reply to the message. Deadlock can be avoided by good design practices or massive kludges — we recommend the good design approach.

device driver

A process that allows the OS and application programs to make use of the underlying hardware in a generic way (e.g. a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino 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.

DNS

Domain Name Service — an Internet protocol used to convert ASCII domain names into IP addresses. In QNX native networking, **dns** is one of **Qnet**'s builtin resolvers.

dynamic bootfile

An OS image built on the fly. Contrast **static bootfile**.

dynamic linking

The process whereby you link your modules in such a way that the Process Manager will link them to the library modules before your program runs. The word “dynamic” here means that the association between your program and the library modules that it uses is done *at load time*, not at linktime. Contrast **static linking**. See also **runtime loading**.

edge-sensitive

One of two ways in which a **PIC** (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is “noticed” upon a transition to/from the rising/falling edge of a pulse. Contrast **level-sensitive**.

edited mode

See **canonical mode**.

EOI

End Of Interrupt — a command that the OS sends to the PIC after processing all Interrupt Service Routines (ISR) for that particular interrupt source so that the PIC can reset the processor's In Service Register. See also **PIC** and **ISR**.

EPROM

Erasable Programmable Read-Only Memory — a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g. 12V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. To change a bit from a 0 state into a 1 state can only be accomplished by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically 10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast **flash** and **RAM**.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requestor of the event.

FD

File Descriptor — a client must open a file descriptor to a resource manager via the *open()* function call. The file descriptor then serves as a handle for the client to use in subsequent messages. Note that a file descriptor is the exact same object as a connection ID (*coid*, returned by *ConnectAttach()*).

FIFO

First In First Out — a scheduling algorithm 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 64k bytes at a time) instead of the entire device. Contrast **EPROM** and **RAM**.

FQNN

Fully Qualified NodeName — 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

Aka 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**.

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).

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

A particular right, that, if enabled for a given thread, allows the thread to perform I/O instructions (such as the x86 assembler `in` and `out` instructions). By default, I/O privileges are disabled, because a program with it enabled can wreak havoc on a system. To enable I/O privileges, the thread must be running as **root**, and call `ThreadCtl()`.

IPC

Interprocess Communication — the ability for two processes (or threads) to communicate. QNX Neutrino 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 `0xFFFFFFFF0` 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`.

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.

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.

Neutrino

Name of an OS developed by QNX Software Systems.

NFS

Network FileSystem — a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g. **ls**, **cp**, **mv**) operate on the remote files exactly as they do on your local files.

NMI

Nonmaskable Interrupt — an interrupt that can't be masked by the processor. We don't recommend using an NMI!

Node Discovery Protocol

See **NDP**.

node domain

A character string that the **Qnet** network manager tacks onto the nodename to form an **FQNN**.

nodename

A unique name consisting of a character string that identifies a node on a network.

nonbootable

A nonbootable OS image is usually provided for larger embedded systems or for small embedded systems where a separate, configuration-dependent setup may be required. Think of it as a second “filesystem” that has some additional files on it. Since it’s nonbootable, it typically won’t contain the OS, startup file, etc. Contrast **bootable**.

OCB

Open Control Block (or Open Context Block) — a block of data established by a resource manager during its handling of the client’s *open()* function. This context block is bound by the resource manager to this particular request, and is then automatically passed to all subsequent I/O functions generated by the client on the file descriptor returned by the client’s *open()*.

package filesystem

A virtual filesystem manager that presents a customized view of a set of files and directories to a client. The “real” files are present on some medium; the package filesystem presents a virtual view of selected files to the client.

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.

Photon microGUI

The proprietary graphical user interface built by QNX Software Systems.

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_ocr_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 QNX Neutrino.

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

Scheduling algorithm 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**.

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**.

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.

software interrupts

Similar to a hardware interrupt (see **interrupt**), except that the source of the interrupt is software.

sporadic

Scheduling algorithm 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.

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 QNX Neutrino. A thread is a flow of execution; it exists within the context of a **process**.

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).



Index

!

“cooked” input mode 222

A

abort() 77
accept() 259
actions (HA) 278
alarm() 94
atomic operations 44
attributes structure (resource
 manager) 178
autoconnect 263
AutoIP 262

B

barriers 20, 44
 and threads 48
bind() 259
bindresvport() 259
block-oriented devices 215

C

canonical input mode 222
cd command 148
CD-ROM
 filesystem 202
CGI (Common Gateway
 Interface) 265
changing directories 148
ChannelCreate() 63
ChannelDestroy() 63
channels 63, 65
character devices 215
chmod() 180
chown() 180
CIFS filesystem 209
clipping (Photon) 316
clock services 90
ClockAdjust() 92
ClockCycles() 92
clock_getcpuclockid() 92
clock_getres() 92
clock_gettime() 92
ClockId() 92
ClockPeriod() 92
clock_settime() 92

ClockTime() 92
close() 84, 180
color model (Photon) 319
combine messages
 defined 177
Common Gateway Interface
 (CGI) 265
conditions (HA entity states) 276
CONDVAR (thread state) 31
condvars 20, 42, 44, 46
 example 47
 operations 46
confstr() 243
connect messages 174
connect() 259
ConnectAttach() 63
ConnectDetach() 63
consoles
 physical 224
 virtual 224
convenience functions (Photon
 widget toolkit) 338
conventions, typographical xvii
cooperating processes
 FIFOs 90
 pipes 89
CRC 196
critical section 45–47, 53, 96
 defined 42
_CS.DOMAIN 244
_CS.HOSTNAME 243

D

data server 265

DEAD (thread state) 31
deadlock-free systems, rules for 67
design goals for QNX Neutrino 3,
 20
devctl() 180, 218
device control 218
device drivers *See also* resource
 managers
 no need to link into kernel 101
 similarity to standard
 processes 13
device names, creating 147
directories
 changing 148
disks
 DOS disks, accessing 199
dladdr() 162
dlclose() 162
dlopen() 161, 162
dlsym() 161
dn_comp() 259
dn_expand() 259
domains of authority 140
DOS filesystem manager 199
dragging windows across network
 links 319
dup() 150, 151, 175, 176
dup2() 150
duplicating file descriptors 150

E

edited input mode 222
editing capabilities (**io-char**) 222
ELF format 133

EMANATE, EMANATE/Lite 264
 embedded web server 264
endprotoent() 259
endservent() 259
 entities (HA process) 275
 ETFS 193
 event space (Photon) 312
 events 69
 “unblock” 70
exclusive (QoS policy) 245
exec()* functions 128, 132
 extensibility of OS 12
 extensions to OS
 user-written, won’t affect
 reliability of core OS 8

F

fast emitting mode (instrumented
 kernel) 109
fcntl() 150
 FIFO (scheduling method) 34, 35,
 44
 FIFOs 90. *See also* pipes
 creating 90
 removing 90
 file descriptors (FDs)
 duplicating 150
 inheritance 150, 151
 open control blocks
 (OCBs) 148
 several FDs referring to the same
 OCB 150
 files
 DOS files, operating on 199

FIFOs 90
 opened by different
 processes 149
 opened twice by same
 process 149
 pipes 89
 filesystem
 accessing a filesystem on
 another node 146
 seek points 149
 transaction 193
 filesystems
 CD-ROM 202
 CIFS 209
 DOS 199
 Image 191
 NFS 208
 QNX 198
 five nines (HA metric) 269
 Flash 133
fork() 128, 131, 132
 POSIX process model and 8
fpathconf() 180
 FQNN (fully qualified node
 name) 244
fseek() 180
fstat() 180
ftruncate() 84

G

gethostbyaddr() 259
gethostbyname() 259
getpeername() 259
getprotobyname() 259

getprotobyname() 259
getprotoent() 259
getservbyname() 259
getservent() 259
getsockname() 259
getsockopt() 259
Global Name Service 243
GNS 243
graphical microkernel 311
graphics drivers 317
 multiple 318
Guardian (HAM “stand-in”) 275

H

HA 269–288
 client-side library 271
 microkernel architecture
 inherently suited for 270
 recovery example 272
HAM 274
 API 283
 hierarchy 275
h_errlist() 259
h_errno() 259
herror() 259
high availability *See* HA
High Availability Manager *See*
 HAM
h_nerr() 259
hsterror() 259
htonl() 259
htons() 259

I

I/O messages 174
I/O resources 140
i8259 interrupt control
 hardware 101
idle thread 33, 103
Image filesystem 191
inet_addr() 259
inet_aton() 259
inet_lnaof() 259
inet_makeaddr() 259
inet_netof() 259
inet_network() 259
inet_ntoa() 259
input
 redirecting 89
input mode
 edited 222
 raw 219
instrumentation
 interrupts can be traced 107
 kernel can be used in prototypes
 or final products 107
 works on SMP systems 107
interprocess communication *See*
 IPC
INTERRUPT (thread state) 31
interrupt control hardware (i8259 on
 a PC) 101
interrupt handlers 20, . *See also*
 ISR 95
interrupt latency 96
Interrupt Service Routine *See* ISR
InterruptAttach() 98
InterruptAttachEvent() 98
InterruptDetach() 98

InterruptDisable() 98
InterruptEnable() 98
InterruptHookIdle() 103
InterruptHookTrace() 103
InterruptLock() 98
InterruptMask() 98
 interrupts
 nested 98
 priorities 98
InterruptUnlock() 98
InterruptUnmask() 98
InterruptWait() 29, 98
intr_timed_wait() 94
io-net 229
ioctl() 259
*iofunc_**(*)* shared library 178
ionotify() 71
 _IO_STAT 170
 IP filtering 261
 IPC 13
 forms of 56
 term qualified to apply to
 “threads” 25
 ISR 99, . *See also* interrupt
 handlers 103
 attaching to PC timer
 interrupt 100

J

JOIN (thread state) 31

K

kill() 72

L

languages 320
 latency
 interrupt 96, 102
 scheduling 97, 102
libpm 292
libpmm 293
libps 292
link() 199
listen() 259
loadbalance (QoS policy) 245
lock() 180
lseek() 180

M

malloc() 155
 memory protection
 advantage of for embedded
 systems 134
 memory, shared 42, 53, 61, 82–89
 message copying 59
 message passing 13, 20, 57
 API 66
 as means of
 synchronization 14
 network-wide 237
 messages
 contents of 14

- multipart 59
- tend to be tiny 42
- microkernel
 - comparable to a realtime executive 8
 - defined 8
 - general responsibilities 11
 - instrumentation 107
 - managing team of cooperating processes 10
 - modularity as key aspect 8
 - services 11
- mkfifo** 90
- mkfifo()* 90
- mmap()* 84, 85, 180
- MMU 60, 61, 135
- mount structure (resource manager) 178
- mountpoint 140
- mprotect()* 84, 89
- mq** server 79
- mq_close()* 80, 81
- mq_getattr()* 81
- mq_notify()* 71, 81
- mq_open()* 80, 81
- mq_receive()* 80, 81
- mq_send()* 80, 81
- mq_setattr()* 81
- mqqueue** resource manager 79
- mq_unlink()* 80, 81
- MsgDeliverEvent()* 30, 66, 70
- MsgError()* 58
- MsgInfo()* 66
- MsgKeyData()* 66
- MsgRead()* 66
- MsgReadv()* 62
- MsgReceive()* 30, 57, 66

- MsgReceivePulse()* 66
- MsgReceivePulsev()* 62
- MsgReceivev()* 62
- MsgReply()* 57, 58, 62, 66
- MsgReply*()* 30
- MsgReplyv()* 62
- MsgSend()* 30, 57, 58, 62, 66
- MsgSendPulse()* 30, 66
- MsgSendsv()* 62
- MsgSendv()* 62, 127
- MsgSendvs()* 62
- MsgWrite()* 66
- msync()* 84
- munmap()* 84
- MUTEX (thread state) 31
- mutexes 20, 42, 44, 45
 - attributes 46
 - priority inheritance 46

N

- name resolution
 - network 240
- NAND flash 193
- NANOSLEEP (thread state) 31
- nanosleep()* 30, 94
- NAT 261
- ND_LOCAL_NODE 240
- nested interrupts 98
- NET_REPLY (thread state) 31
- NET_SEND (thread state) 31
- network
 - as homogeneous set of resources 14

- Driver Development Kit
 - (DDK) 233
- flexibility 15
- message passing 237
- name resolution 240
- pathnames 148
- protocol
 - Qnet 237
- transparency 15, 172
- Neutrino
 - microkernel 8
- NFS filesystem 208
- NMI 103
- node descriptor
 - network 240
- non-Ethernet interconnect 251
- npm-qnet** 244
- npm-tcpip** 257
- ntohl()* 259
- ntohs()* 259
- NTP 262

O

- opaque bitmask (Photon) 314
- open control blocks (OCBs) 148, 149
- open resources
 - active information contained in
 - OCBs 149
- open()* 84, 180
- output, redirecting 89

P

- parallel devices 225
- pathconf()* 180
- pathname
 - converting relative to
 - network 148
- pathname delimiter in QNX
 - Momentics documentation
 - xviii
- pathname space 140
 - mapping 167
- pause()* 72
- performance
 - context-switch 43
 - realtime 95
- persistent storage 304
- Photon
 - architecture differs from X
 - Window System 311
 - architecture similar to that of OS
 - microkernel 311
 - event space 312
 - event types 316
 - graphics drivers 317, 319
 - microkernel runs as tiny
 - process 311
 - regions 314
 - widgets 325–342
 - window manager 324
- pipe** manager 89
- pipe()* 90
- pipes 89
 - creating 90
- Point-to-Point Protocol (PPP) 257
- Point-to-Point Protocol over Ethernet (PPPoE) 263

- popen()* 90
- POSIX
 - defines interface, not implementation 3
 - profiles 5
 - realtime extensions 5
 - standards of interest to
 - embedded systems developers 4
 - suitable for embedded systems 4, 6
 - threads 5
 - library calls not involving kernel calls 23
 - library calls with corresponding kernel calls 24
 - UNIX and 3
- power management 291–308
 - CPU modes 306
 - framework components 291
 - in embedded systems 291
 - industry standards are
 - PC-oriented 291
- power manager 294
 - components of 295
 - is written by the user 295
- power policy 295, 296
 - determined by the developer 291
- power states 297
- power-aware applications 304
- power-managed objects 299
- PPP (Point-to-Point Protocol) 257
- PPPoE (Point-to-Point Protocol over Ethernet) 263
- preferred** (QoS policy) 245
- prefix 140
- printf()* 155
- priority 32
 - inheritance 131
 - inversion 34, 46
 - range 33
- process
 - as container for threads 23
 - loading 133
 - management 127
 - model
 - required for POSIX compliance 20
 - primitives 128
- process groups
 - membership, inheriting 130
 - remote node 131
- Process Manager
 - capabilities of 127
 - required when creating multiple POSIX processes 127
- processes
 - cooperating
 - via pipes and FIFOs 89
 - opening the same file twice 149
 - OS as team of cooperating 10
 - system 12
- product line, using a single OS
 - for 4
- PROT.EXEC 86
- PROT.NOCACHE 86
- PROT.NONE 86
- PROT.READ 86
- PROT.WRITE 86
- pthread_attr_destroy()* 24
- pthread_attr_getdetachstate()* 24

- pthread_attr_getinheritsched()* 24
 - pthread_attr_getschedparam()* 24
 - pthread_attr_getschedpolicy()* 24
 - pthread_attr_getscope()* 24
 - pthread_attr_getstackaddr()* 24
 - pthread_attr_getstacksize()* 24
 - pthread_attr_init()* 24
 - pthread_attr_setdetachstate()* 24
 - pthread_attr_setinheritsched()* 24
 - pthread_attr_setschedparam()* 24
 - pthread_attr_setschedpolicy()* 24
 - pthread_attr_setscope()* 24
 - pthread_attr_setstackaddr()* 24
 - pthread_attr_setstacksize()* 24
 - pthread_barrierattr_destroy()* 50
 - pthread_barrierattr_getpshared()* 50
 - pthread_barrierattr_init()* 50
 - pthread_barrierattr_setpshared()* 50
 - pthread_barrier_destroy()* 50
 - pthread_barrier_init()* 48
 - pthread_barrier_wait()* 48
 - pthread_cancel()* 24
 - pthread_cleanup_pop()* 24
 - pthread_cleanup_push()* 24
 - pthread_cond_broadcast()* 24, 55
 - pthread_cond_destroy()* 24, 55
 - pthread_cond_init()* 24, 55
 - pthread_cond_signal()* 24, 55
 - pthread_cond_timedwait()* 55, 94
 - pthread_condvar_wait()* 29
 - pthread_cond_wait()* 24, 55
 - pthread_create()* 24
 - pthread_detach()* 24
 - pthread_equal()* 24
 - pthread_exit()* 24
 - pthread_getcpuclockid()* 92
 - pthread_getschedparam()* 24
 - pthread_getspecific()* 24, 27
 - pthread_join()* 24, 29
 - pthread_key_create()* 24, 27
 - pthread_key_delete()* 24
 - pthread_kill()* 24, 72
 - pthread_mutex_destroy()* 24, 55
 - pthread_mutex_init()* 24, 55
 - pthread_mutex_lock()* 24, 30, 55
 - pthread_mutex_setrecursive()* 46
 - pthread_mutex_trylock()* 24, 55, 94
 - pthread_mutex_unlock()* 24, 55
 - pthread_once()* 24
 - pthread_rwlock_rdlock()* 51
 - pthread_rwlock_tryrdlock()* 51
 - pthread_rwlock_trywrlock()* 51
 - pthread_rwlock_unlock()* 51
 - pthread_rwlock_wrlock()* 51
 - pthread_self()* 24
 - pthread_setcancelstate()* 24
 - pthread_setcanceltype()* 24
 - pthread_setschedparam()* 24
 - pthread_setspecific()* 24
 - pthread_sigmask()* 24
 - pthread_sleepon_lock()* 51
 - pthread_testcancel()* 24
 - pty (pseudo terminal)
 - as pair of character devices 226
 - pulses 65, 70
- Q**
- Qnet 232, 237–251

QNX Neutrino
 design goals 3, 20
 extensibility 12
 flexibility 7
 network as homogeneous set of
 resources 14
 network flexibility 15
 network transparency 15
 preemptable even during
 message pass 21
 realtime applications, suitability
 for 7
 services 20
 single-computer model 15
QoS (Quality of Service) 244
 policies 245

R

raise() 72
RAM 133
RAM disk 187
RapidIO 251
rate monotonic analysis (RMA) 37
raw input mode 219
 conditions for input
 request 220
 FORWARD qualifier 221
 MIN qualifier 220
 TIME qualifier 220
 TIMEOUT qualifier 220
read() 71
readblock() 177
readdir() 145, 184
reader/writer locks 44, 51

READY (thread state) 31
realtime performance 95
 interrupt latency and 96
 nested interrupts and 98
 scheduling latency and 97
RECEIVE (thread state) 31
rectangle list (Photon) 316
recv() 259
recvfrom() 259
redirecting
 input 89
 output 89
regions (Photon) 312, 314
relative pathnames, converting to
 network pathnames 148
remove() 90
REPLY (thread state) 31
res_init() 259
res_mkquery() 259
resource managers
 atomic operations 177
 attributes structure 178
 can be started and stopped
 dynamically 167
 communicate with clients via
 IPC 172
 context for client requests 174
 defined 167
 don't require any special
 arrangements with the
 kernel 167
 iofunc_()* shared library 178
 message types 174
 mount structure 178
 shared library 174
 similarity to traditional device
 drivers 167

- similarity to user-level
 - servers 167
 - thread management 176
 - unique features of 172
- resources
 - accessing on other
 - machines 15
 - no real difference between local
 - and remote 14
 - open 149
- res_query()* 259
- res_querydomain()* 259
- res_search()* 259
- res_send()* 259
- rm** 90
- robustness
 - improved via memory
 - protection 134
 - of application architectures via
 - Send/Receive/Reply 67
 - of implementations with
 - Send/Receive/Reply 67
 - of IPC 56
 - of signal handlers 70
- ROM 133
- round-robin scheduling 34, 36
- RUNNING (thread state) 31

S

- scaling
 - advantages of 4
 - of applications 4
- scatter/gather DMA 59
- SchedGet()* 24, 41

- sched_getparam()* 41
- sched_getscheduler()* 41
- SchedSet()* 24, 41
- sched_setparam()* 41
- sched_setscheduler()* 41
- scheduling
 - FIFO 34, 35
 - latency 97
 - method
 - determining 35
 - setting 35
 - round-robin 34, 36
 - sporadic 34, 37
 - threads 32
- sched_yield()* 32
- SCTP 260
- seek points 149, 151
- select()* 71, 259
- SEM (thread state) 31
- semaphores 20, 42, 44, 52
 - named 52
- sem_destroy()* 55
- sem_init()* 55
- sem_post()* 55
- sem_trywait()* 55
- sem_wait()* 55
- SEND (thread state) 31
- send()* 259
- sendto()* 259
- sensitivity bitmask (Photon) 314
- serial
 - devices 225
 - drivers 225
 - tiny versions 225
- Server Side Includes (SSI) 265
- services, handled by system
 - processes 12

- sessions
 - remote node 131
- setprotoent()* 259
- setservent()* 259
- setsockopt()* 259
- shared libraries (*.so* files) 19, 127
- shared memory 42, 53, 61, 82–89
- shm_ctl()* 84
- shm_open()* 84
- shm_unlink()* 84
- shutdown()* 259
- SIGABRT 77
- sigaction()* 72
- SIGALRM 77
- SIGBUS 77
- SIGCHLD 77
- SIGCONT 77
- SIGDEADLK 77
- SIGEMT 77
- SIGFPE 77
- SIGHUP 77
- SIGILL 77
- SIGINT 77
- SIGIOT 77
- SIGKILL 77
- signal()* 75
- SignalAction()* 72
- SignalKill()* 24, 30, 72
- SignalProcMask()* 24
- SignalProcmask()* 72
- signals 20
 - POSIX and UNIX 72
 - queuing of 74
 - rules for a multi-threaded
 - process 73
 - similarity to pulses 74
 - targeted at specific threads 72
- SignalSuspend()* 72
- SignalWaitinfo()* 72
- SIGPIPE 77
- SIGPOLL 77
- sigprocmask()* 72
- sigqueue()* 72
- SIGQUIT 77
- SIGSEGV 77
- SIGSTOP 77
- SIGSUSPEND (thread state) 31
- sigsuspend()* 30, 72
- SIGSYS 77
- SIGTERM 77
- sigtimedwait()* 94
- SIGTRAP 77
- SIGTSTP 77
- SIGTTIN 77
- SIGTTOU 77
- SIGURG 77
- SIGUSR1 77
- SIGUSR2 77
- SIGWAITINFO (thread state) 31
- sigwaitinfo()* 30, 72
- SIGWINCH 77
- single-computer model 15
- sleep()* 94
- sleep on locks 44, 51
- slinger** 264
- SNMP 264
- socket()* 259
- software interrupt *See* signals
- spawn()*
 - family of functions 128, 129
 - QNX implementation 129
- SPAWN_SETGROUP 130
- SPAWN_SETSIGDEF 130
- SPAWN_SETSIGMASK 130

SPOF 274
 sporadic scheduling 34, 37
 SSI (Server Side Includes) 265
 STACK (thread state) 31
stat() 180
 states
 CONDVAR 31
 DEAD 31
 INTERRUPT 31
 JOIN 31
 MUTEX 31
 NANOSLEEP 31
 NET_REPLY 31
 NET_SEND 31
 READY 31
 RECEIVE 31
 REPLY 31
 RUNNING 31
 SEM 31
 SEND 31
 SIGSUSPEND 31
 SIGWAITINFO 31
 STACK 31
 STOPPED 31
 WAITCTX 31
 WAITPAGE 31
 WAITTHREAD 31
 STOPPED (thread state) 31
stty 223
 symbolic links
 cd command and 148
 symbolic prefixes 145
SyncCondvarSignal() 24, 55
SyncCondvarWait() 24, 55
SyncDestroy() 24, 55
 synchronization services 44, 55
SyncMutexEvent() 24

SyncMutexLock() 24, 55
SyncMutexUnlock() 24, 55
SyncSemPost() 55
SyncSemWait() 30, 55
SyncTypeCreate() 24, 55
 system processes 12
 similarity to user-written
 processes 12

T

tcdropline() 218
tcgetattr() 218
tcgetpgrp() 218
tcinject() 218
 TCP/IP 255–265
 resource manager
 (**npm-tcpip**) 257
 stack configurations 255
tcsendbreak() 218
tcsetattr() 218
tcsetpgrp() 218
 terminal emulation 225
textto 199
ThreadCancel() 24
ThreadCreate() 24, 31
ThreadCtl() 24
ThreadDestroy() 24
ThreadDetach() 24
ThreadJoin() 24
 threads 20, 176
 all share same kernel
 interface 127
 and barriers 48
 attributes of 26

- cancellation handlers 26
- concurrency advantages 43
- defined 23
- life cycle 28
- priority inversion 34
- process must contain one or more 23
- register set 26
- scheduling 31
 - FIFO 35
 - round-robin 36
 - sporadic 37
- signal mask 26
- stack 26
- states 28
- synchronization 42
- tid* 26
- TLS (thread local storage) 26
- timeout service 93
- TimerAlarm()* 94
- TimerCreate()* 94
- timer_create()* 94
- timer_delete()* 94
- TimerDestroy()* 94
- TimerGetoverrun()* 94
- timer_getoverrun()* 94
- TimerGettime()* 94
- timer_gettime()* 94
- timers 20, 93
 - cyclical mode 93
- TimerSettime()* 94
- timer_settime()* 94
- TimerTimeout()* 94
- timeslice 36
- TLB (translation look-aside buffer) 135
- TLS (thread local storage) 26

- transparency of network 15
- typographical conventions xvii

U

- UART 102
- Unicode 320
- unlink()* 90
- UTF-8 encoding 321
- utime()* 180

V

- vfork()* 128, 132
- virtual addresses 135
- virtual consoles 224

W

- WAITCTX (thread state) 31
- WAITPAGE (thread state) 31
- WAITTHREAD (thread state) 31
- watchdog 136
- web server 264
- wide emitting mode (instrumented kernel) 109
- widgets (Photon) 325–342
- window manager 324
- write()* 71, 155

Z

zero-copy architecture
 (**io-net**) 230