

QNX® 4 Operating System

System Architecture

QNX Software Systems Ltd.

175 Terence Matthews Crescent
Kanata, Ontario K2M 1W8
Canada
613-591-0931

© QNX Software Systems Ltd. 1993
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of QNX Software Systems Ltd.

Although every precaution has been taken in the preparation of this book, we assume no responsibility for any errors or omissions, nor do we assume liability for damages resulting from the use of the information contained in this book.

Printing history:

January 1992	First edition
January 1993	Reprint
July 1993	Minor changes
January 1994	Reprint
April 1994	Reprint
September 1994	Reprint
February 1995	Reprint

QNX and the  symbol are registered trademarks of QNX Software Systems Ltd.
All other trademarks and registered trademarks belong to their respective owners.

Cover art by Robert Tinney.

Printed in Canada.

Acknowledgments

Several members of QNX Software Systems Ltd. have contributed to the design and implementation of QNX 4. This product embodies more than a decade of research, development, and field-tested experience. We are proud to see the QNX operating system enter the standards arena without sacrificing the fundamental qualities of its original vision.

The following people deserve a special word of thanks and acknowledgment for their substantial contribution to QNX 4:

Software Development:

Luc N. Bazinet
Gordon Bell
Andrew C. Boyd
Robin Burgener
Brian Campbell
Grant D. Courville
Dan Dodge
Andrew Edgar
William A. Flowers
Dan Hildebrand
Eric Johnson
Steve McPolin
Rob Oakley
Peter van der Veen
Martin Walter

Documentation:

Kevin Didsbury
Paul N. Leroux
John M. Ostrander

A Word about POSIX

QNX 4.1 follows the Institute of Electrical and Electronics Engineers (IEEE) POSIX standards (the term “POSIX” stands for *portable operating system interface* and refers collectively to an entire group of related standards).

POSIX is intended to address the need for a common denominator among the myriad flavors of UNIX operating systems on the market. Sanctioned by several internationally recognized standards bodies and supported by numerous industry participants, the POSIX standard serves as a verifiable safeguard of portability.

The POSIX standard clearly intends to define only the “external characteristics” (POSIX Std 1003.1-1988 Scope §1.), which is why various operating systems may look and behave in much the same way “on the surface,” while their performance, internal architecture, and other distinctive features may differ dramatically.

For more information on POSIX or to obtain copies of published POSIX standards, contact the IEEE:

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08855-1331
Tel (toll-free): 800-678-IEEE
Tel: 201-981-0060

What will I find in this book?

The *System Architecture* book accompanies the QNX operating system and is intended for both application developers and end-users.

This book provides a detailed look at the structure and functions of QNX. It describes the Microkernel, System Managers, as well as QNX's unique message-based IPC. We recommend that you read this book before you begin using QNX.

For information on installing and using QNX, refer to the *User's Guide*.

The *System Architecture* book contains the following chapters:

- The Philosophy of QNX
- The Microkernel
- The Process Manager
- I/O Namespace
- The Filesystem Manager
- The Device Manager
- The Network Manager

Conventions used in this book

Throughout this book, 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
command names	ls
data types	<i>long</i>
environment variables	PATH
error numbers	[EINTR]
filenames	/dev/tty
functions	<i>system()</i>
function arguments	<i>arg1</i>
global externals	<i>errno</i>
headers	<sys/stat.h>
options	-c
option arguments	-w <i>width</i>
operands	<i>file_name</i>
parameters	<directory pathname>
symbolic constants	{_POSIX_VDISABLE}

Contents

Chapter 1 The Philosophy of QNX 1

- What is QNX? 2
- QNX's microkernel architecture 3
 - A true kernel 3
 - System processes 4
- Interprocess communication 5
 - A message-passing operating system 5
- QNX as a network 6
 - The single-computer model 7
 - Flexible networking 7

Chapter 2 The Microkernel 9

- Introduction 10
- Interprocess communication (IPC) 11
- IPC via messages 11
 - Message-passing primitives 11
 - Process synchronization 13
 - Blocked states 13
 - Using *Send()*, *Receive()*, and *Reply()* 14
 - Reply-driven messaging 16
 - Additional points to remember 16
 - Advanced facilities 18
 - Reserved message codes 20
- IPC via proxies 21

The Microkernel (cont'd)

IPC via signals	22
Generating signals	22
Receiving signals	23
Summary of signals	24
Defining signal processing	26
Catching signals	26
Safe functions for signal handlers	26
Blocking signals	27
Signals and messages	28
IPC across the network	29
Virtual circuits	29
Virtual processes	29
Terminating virtual circuits	31
Process scheduling	32
When scheduling decisions are made	32
Process priorities	32
Scheduling methods	33
A word about realtime performance	37
Interrupt latency	37
Scheduling latency	38
Stacked interrupts	39

Chapter 3 The Process Manager 41

Introduction	42
Process Manager responsibilities	42
Process creation primitives	42
Process inheritance	43
The life cycle of a process	44
Creation	44
Loading	44
Execution	44
Termination	44

Process states	46
Determining process states	48
Process symbolic names	48
Timers	49
Time management	49
Simple timing facilities	49
Advanced timing facilities	50
Interrupt handlers	51
Timer interrupt handlers	52

Chapter 4 **I/O Namespace** **53**

Introduction	54
I/O namespace	54
Prefixes and regions of authority	54
Resolving pathnames	54
I/O manager prefixes	54
Network root	55
Default network root	56
Alias prefixes	58
Relative pathnames	59
Current working directory	59
File descriptor namespace	61
Open control blocks	61

Chapter 5 **The Filesystem Manager** **65**

Introduction	66
What is a file?	66
Date and time stamps	66
File access	67
Regular files and directories	67
Regular files	67
Directories	67
Extents	69
Links and inodes	70
Symbolic links	72

The Filesystem Manager (cont'd)

Pipes and FIFOs	74
Pipes	74
FIFOs	75
Filesystem Manager performance	75
Elevator seeking	75
Buffer cache	75
Multi-threading	76
Client-driven priority	77
Temporary files	77
Ramdisks	77
Filesystem robustness	77
Filesystem recovery	78
Raw volumes	78
Disks and disk subsystems	78
OS partitions	79
Defining block special files	81
Mounting a filesystem	82
Unmounting a filesystem	82
Key components of a QNX partition	83
Loader block	83
Root block	84
Bitmap	84
Root directory	84
The DOS Filesystem Manager	85

Chapter 6 The Device Manager 87

Introduction	88
Device services	88
Edited input mode	89
Raw input mode	91
Device drivers	92
Device control	94

The QNX console	94
Console-specific functions	95
Serial devices	95
Parallel devices	96
Device subsystem performance	97

***Chapter 7* The Network Manager 99**

Introduction	100
Network Manager responsibilities	100
An independent module	100
Microkernel/Network Manager interface	101
Network drivers	104
Node and network identifiers	104
Physical node ID	104
Logical node ID	105
Logical network ID	105
Choosing a network	106
Load balancing	106
Fault tolerance	107

Index 109

Chapter 1

The Philosophy of QNX

In this Chapter...

What is QNX?

QNX's microkernel architecture

Interprocess communication

QNX as a network

What is QNX?

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 operating system to handle multiple events within fixed time constraints. The more responsive the OS, the more “room” a realtime application has to maneuver when meeting its deadlines.

The QNX Operating System is ideal for realtime applications. It provides multitasking, priority-driven preemptive scheduling, and fast context switching—all essential ingredients of a realtime system.

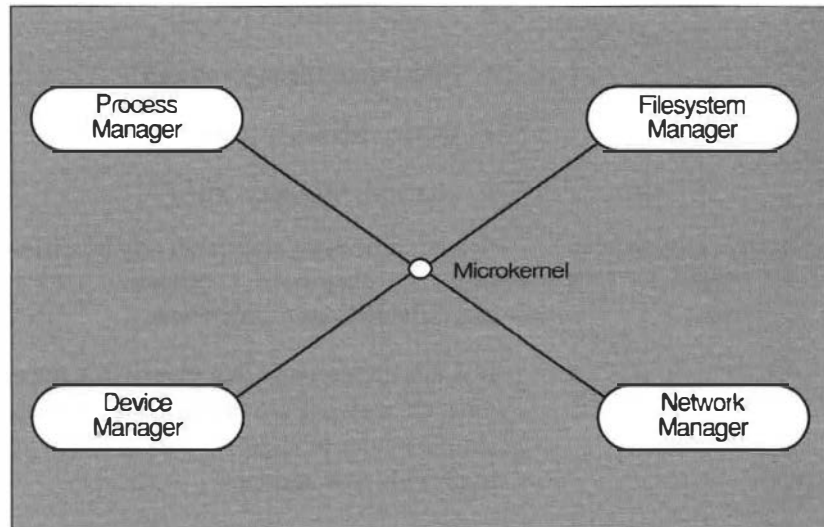
QNX is also remarkably flexible. Developers can easily customize the operating system to meet the needs of their application. From a “bare-bones” configuration of a kernel with a few small modules to a full-blown network-wide system equipped to serve hundreds of users, QNX lets you set up your system to use only those resources you require to tackle the job at hand.

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

- microkernel architecture
- message-based interprocess communication

QNX's microkernel architecture

QNX consists of a small kernel in charge of 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 and with their “quarterback” kernel.



The QNX Microkernel coordinating the system managers.

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 the QNX Kernel is truly a kernel. First of all, like the kernel of a realtime executive, the QNX Kernel is very small—less than 8K. Secondly, it’s dedicated to only two essential functions:

- **message passing**—the Kernel handles the routing of all messages among all processes throughout the entire system
- **scheduling**—the scheduler is a part of the Kernel, and is invoked whenever a process changes state as the result of a message or interrupt

Unlike processes, the **Kernel** itself is never scheduled for execution. It is entered only as the direct result of Kernel calls, either from a process or from a hardware interrupt.

System processes

All QNX services, except those provided by the Kernel, are handled via standard QNX processes. A typical QNX configuration has the following system processes:

- **Process Manager (Proc)**
- **Filesystem Manager (Fsys)**
- **Device Manager (Dev)**
- **Network Manager (Net)**

System processes vs.
user-written processes

System processes are practically no different from any user-written program—they have no private or hidden interfaces that are unavailable to user processes.

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

Let's suppose you've written a database server. How should such a process be classified?

Just as a filesystem accepts requests (messages in QNX) 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 a set of primitives (implemented by messages) which in turn provide access to a resource. Both are independent processes that can be written by an end-user and started 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 QNX allows such processes to be implemented cleanly, with no need at all for modifications to the standard components of the operating system.

Device drivers

Device drivers are processes that shield the operating system from dealing with all the details required for supporting specific hardware.

Since drivers start up as standard processes, adding a new driver to QNX doesn't affect any other part of the operating system. The only change you need to make to your QNX environment is to actually start the new driver.

Once they've completed their initialization, drivers can do either of the following:

- choose to disappear as standard processes, simply becoming extensions to the system process they're associated with
- retain their individual identity as standard processes

Interprocess communication

When several processes run concurrently, as in typical realtime multitasking environments, the operating system must provide mechanisms to allow processes 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.

QNX provides a simple but powerful set of IPC capabilities that greatly simplify the job of developing applications made up of cooperating processes.

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. QNX owes much of its power, simplicity, and elegance to the complete integration of the message-passing method throughout the entire system.

In QNX, a message is a packet of bytes passed from one process to another. QNX 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 Kernel 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’s message-passing design helps bring order and greater reliability to applications.

QNX as a network

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

Any process on any machine in the network can directly make use of any resource on any other machine. From the application’s perspective, there is no difference between a local or remote resource and no special facilities need to be built into applications to make use of remote resources. In fact, a program would need special code to be able to tell whether a resource such as a file or device was present on the local computer or was on some other node on the network!

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, QNX's all-pervasive message-passing IPC accounts for such fluid, transparent networking.

The single-computer model

QNX is designed from the ground up as a network-wide operating system. In some ways, a QNX network feels more like a mainframe computer than a set of micros. Users are simply aware of a large set of resources available for use by any application. But unlike a mainframe, QNX 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 process control environment, for example, PLCs and other realtime I/O devices may require more resources than other, less critical, applications, such as a word processor. The QNX network is responsive enough to support both types of applications *at the same time*—QNX lets you focus computing power on the plant floor where and when it's needed, without sacrificing concurrent connectivity to the desktop.

Flexible networking

QNX 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 operating system.



The list of specific network hardware that QNX supports may grow over time. For details, consult the documentation for the network hardware you will be using.

Each node in a QNX network is assigned a unique number that becomes its identifier. This number is the only visible means to determine whether QNX is running as a network or as a single-processor operating system.

This degree of transparency is yet another example of the distinctive power of QNX'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!

QNX, on the other hand, is grounded on the principle that effective communication is the key to effective operation. Message passing thus forms the cornerstone of QNX’s 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 coax.

Let’s now take a closer look at the structure and functions of QNX.

Chapter 2

The Microkernel

In this Chapter...

Introduction

Interprocess communication (IPC)

IPC via messages

IPC via proxies

IPC via signals

IPC across the network

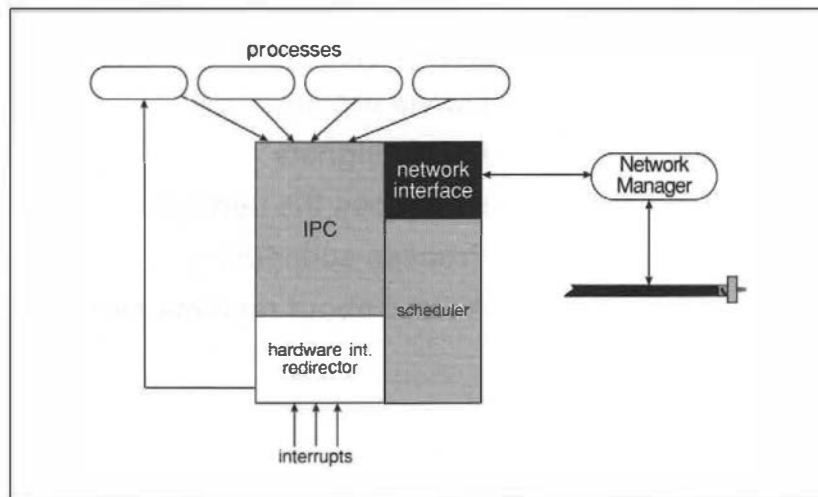
Process scheduling

A word about realtime performance

Introduction

The QNX Kernel is responsible for the following:

- **interprocess communication**—the Kernel supervises the routing of *messages*; it also manages two other forms of IPC—*proxies* and *signals*
- **low-level network communication**—the Kernel delivers all messages destined for processes on other nodes
- **process scheduling**—the Kernel's *scheduler* decides which process will execute next
- **first-level interrupt handling**—all hardware interrupts and faults are first routed through the Kernel, then passed on to the appropriate driver or system manager



The QNX Microkernel.

Interprocess communication (IPC)

The QNX Kernel supports three essential types of interprocess communication: messages, proxies, and signals.

Messages are the fundamental form of IPC in QNX. They provide synchronous communication between cooperating processes where the process sending the message requires proof of receipt and potentially a reply to the message.

Proxies are a special form of message. They're especially suited for event notification where the sending process doesn't need to interact with the recipient.

Signals are a traditional form of IPC. They're used to support asynchronous interprocess communication.

IPC via messages

In QNX, a message is a packet of bytes that's synchronously transmitted from one process to another. QNX attaches no meaning to the content of a message. The data in a message has meaning for the sender and for the recipient, but for no one else.

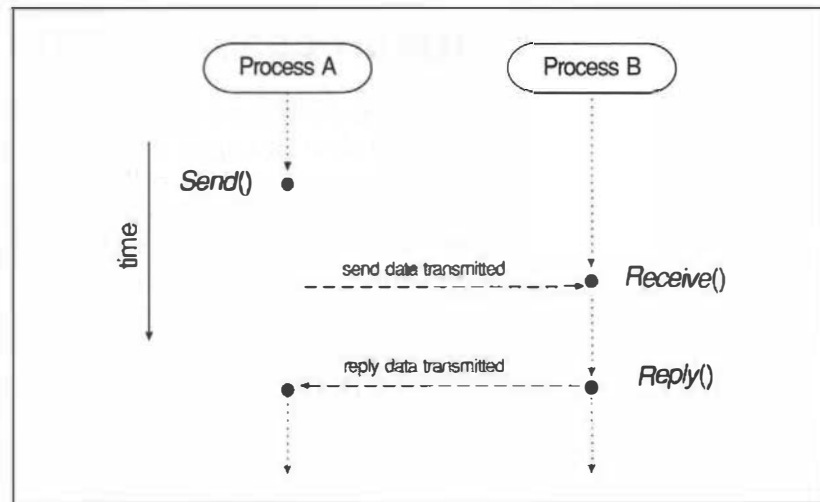
Message-passing primitives

To communicate directly with one another, cooperating processes use these C language functions:

- *Send()*—for sending messages
- *Receive()*—for receiving messages
- *Reply()*—for replying to processes that have sent messages

These functions may be used locally or across the network.

Note also that unless processes want to communicate directly with each other, they don't need to use *Send()*, *Receive()*, and *Reply()*. The QNX C Library is built on top of messaging—processes use messaging indirectly when they use standard services, such as pipes.



Process A sends a message to Process B which subsequently receives, processes, then replies to the message.

The above illustration outlines a simple sequence of events in which two processes, Process A and Process B, use *Send()*, *Receive()*, and *Reply()* to communicate with each other:

- 1) Process A sends a message to Process B by issuing a *Send()* request to the kernel. At this point, Process A becomes SEND-blocked until Process B issues a *Receive()* to receive the message.
- 2) Process B issues a *Receive()* and receives Process A's waiting message. Process A changes to a REPLY-blocked state. Since a message was waiting, Process B doesn't block.

(Note that if Process B had issued the *Receive()* before a message was sent, it would become RECEIVE-blocked until a message arrived. In this case, the sender would immediately go into the REPLY-blocked state when it sent its message.)

- 3) Process B completes the processing associated with the message it received from Process A and issues a *Reply()*. The reply message is copied to Process A, which is made ready to run. A *Reply()* doesn't block, so Process B is also ready to run. Who runs depends on the relative priorities of Process A and Process B.

Process synchronization

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several cooperating processes.

Let's look at the above illustration again. Once Process A issues a *Send()* request, it's unable to resume execution until it has received the reply to the message it sent. This ensures that the processing performed by Process B for Process A is complete before Process A can resume executing. Moreover, once Process B has issued its *Receive()* request, it can't continue processing until it receives another message.



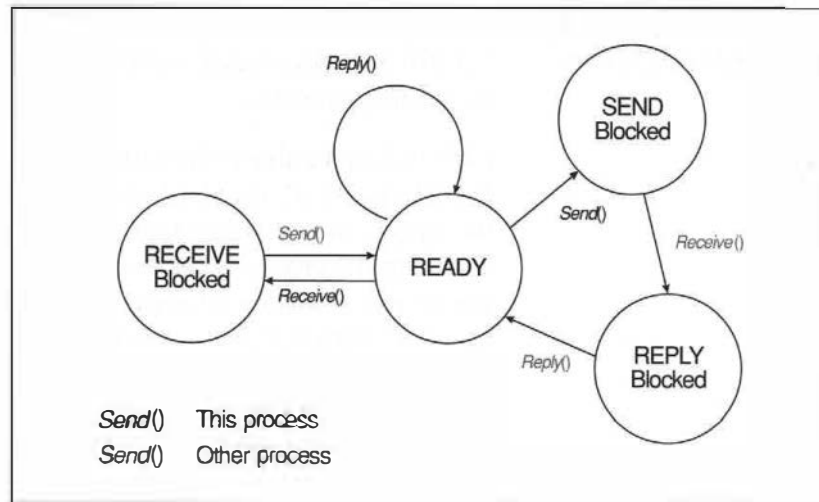
For details on how QNX schedules processes, see “Process scheduling” in this chapter.

Blocked states

When a process isn't allowed to continue executing—because it must wait for some part of the message protocol to end—the process is said to be *blocked*.

The following table summarizes the blocked states of processes:

If a process has issued a:	the process is:
<i>Send()</i> request, and the message it has sent hasn't yet been received by the recipient process	SEND-blocked
<i>Send()</i> request, and the message has been received by the recipient process, but that process hasn't yet replied	REPLY-blocked
<i>Receive()</i> request, but hasn't yet received a message	RECEIVE-blocked



States involved in a typical send-receive-reply transaction.



For information on all possible process states, see Chapter 3.

Using *Send()*, *Receive()*, and *Reply()*

Send()

Let's now take a closer look at the *Send()*, *Receive()*, and *Reply()* function calls. We'll stay with our example of Process A and Process B.

Let's assume Process A issues a request to send a message to Process B. It issues the request by means of a *Send()* function call:

```
Send ( pid, msg, rmsg, msg_len, rmsg_len );
```

The *Send()* call contains these arguments:

<i>pid</i>	the <i>process ID</i> of the process that is to receive the message (i.e. Process B); a <i>pid</i> is the identifier by which the process is known to the operating system and to other processes
<i>msg</i>	the message buffer (i.e. the message to be sent)
<i>rmsg</i>	the reply buffer (will contain the reply from Process B)
<i>msg_len</i>	the length of the message being sent
<i>rmsg_len</i>	the maximum length of the reply that Process A will accept

Note that no more than *smsg_len* bytes will be sent, and no more than *rmsg_len* bytes will be accepted in the reply—this ensures that buffers aren't accidentally overwritten.

Receive() Process B can receive the *Send()* issued from Process A by issuing a *Receive()* call:

```
pid = Receive( 0, msg, msg_len );
```

The *Receive()* call contains these arguments:

<i>pid</i>	the process ID of the process that sent the message (i.e. Process A) is returned
0	(zero) specifies that Process B is willing to accept a message from any process
<i>msg</i>	the buffer where the message will be received
<i>msg_len</i>	the maximum amount of data that will be accepted in the receive buffer

If the *smsg_len* in the *Send()* call and the *msg_len* in the *Receive()* call differ in size, the smaller of the two determines the amount of data that will be transferred.

Reply() Having successfully received the message from Process A, Process B should reply to Process A by issuing a *Reply()* function call:

```
Reply( pid, reply, reply_len );
```

The *Reply()* call contains these arguments:

<i>pid</i>	the process ID of the process to which the reply is directed (i.e. Process A)
<i>reply</i>	the reply buffer
<i>reply_len</i>	the length of the data to be transmitted in the reply

If the *reply_len* in the *Reply()* call and the *rmsg_len* in the *Send()* call differ in size, the smaller of the two determines how much data will be transferred.

Reply-driven messaging

The messaging example we just looked at illustrates the most common use of messaging—that in which a server process is normally RECEIVE-blocked for a request from a client in order to perform some task. This is called *send-driven messaging*: the client process initiates the action by sending a message, and the action is finished by the server replying to the message.

Although not as common as send-driven messaging, another form of messaging is also possible—and often desirable—to use:

reply-driven messaging, in which the action is initiated with a *Reply()* instead. Under this method, a “worker” process sends a message to the server indicating that it’s available for work. The server doesn’t reply immediately, but rather “remembers” that the worker has sent an arming message. At some future time, the server may decide to initiate some action by replying to the available worker process. The worker process will do the work, then finish the action by sending a message containing the results to the server.

Additional points to remember

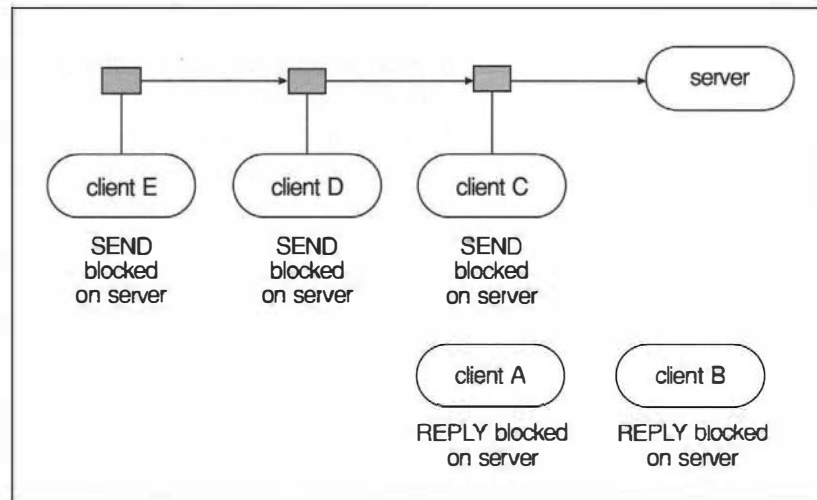
Here are some more things to keep in mind about message passing:

- The message data is maintained in the sending process until the receiver is ready to process the message. There is *no* copying of the message into the Kernel. This is safe since the sending process is SEND-blocked and is unable to inadvertently modify the message data.
- The message reply data is copied from the replying process to the REPLY-blocked process as an atomic operation when the *Reply()* request is issued. The *Reply()* doesn’t block the replying process—the REPLY-blocked process becomes unblocked after the data is copied into its space.
- The sending process doesn’t need to know anything about the state of the receiving process before sending a message. If the receiving process isn’t prepared to receive a message when the sending process issues it, the sending process simply becomes SEND-blocked.
- If necessary, a process can send a zero-length message, a zero-length reply, or both.

- From the developer's point of view, issuing a *Send()* to a server process to get a service is virtually identical to calling a library subroutine to get the same service. In either case, you set up some data structures, then make the *Send()* or the library call. All of the service code between two well-defined points—*Receive()* and *Reply()* for a server process, function entry and **return** statement for a library call—then executes while your code waits. When the service call returns, your code “knows” where results are stored and can proceed to check for error conditions, process results, or whatever.

Despite this apparent simplicity, the code does much more than a simple library call. The *Send()* may transparently go across the network to another machine where the service code actually executes. It can also exploit parallel processing without the overhead of creating a new process. The server process can issue a *Reply()*, allowing the caller to resume execution as soon as it is safe to do so, and meanwhile continue its own execution.

- There may be messages outstanding from many processes for a single receiving process. Normally, the receiving process receives the messages in the order they were sent by other processes; however, the receiving process can specify that messages be received in an order based on the priority of the sending processes.



Server has received (but not replied to) messages from Client A and Client B. Messages from Client C, Client D, and Client E have not yet been received.

Advanced facilities

QNX also provides these advanced message-passing facilities:

- conditional message reception
- reading or writing part of a message
- multi-part messages

Conditional message reception

Generally, when a process wants to receive messages, it uses *Receive()* to wait for a message to arrive. This is the normal way of receiving messages and is appropriate in most circumstances.

In some cases, however, a process may need to determine whether messages are pending, yet may not want to become *RECEIVE*-blocked in the absence of a pending message. For example, a process needs to poll a free-running device at high speed—the device isn't capable of generating interrupts—but the process still has to respond to messages from other processes. In this case, the process could use the *Creceive()* function to read a message, if one became available, yet return immediately if no further messages were pending.



You should avoid *Creceive()*, if possible, since it allows a process to consume the processor continuously at its priority level.

Reading or writing part of a message

Sometimes it's desirable to read or write only part of a message at a time so that you can use the buffer space already allocated for the message instead of allocating a separate work buffer.

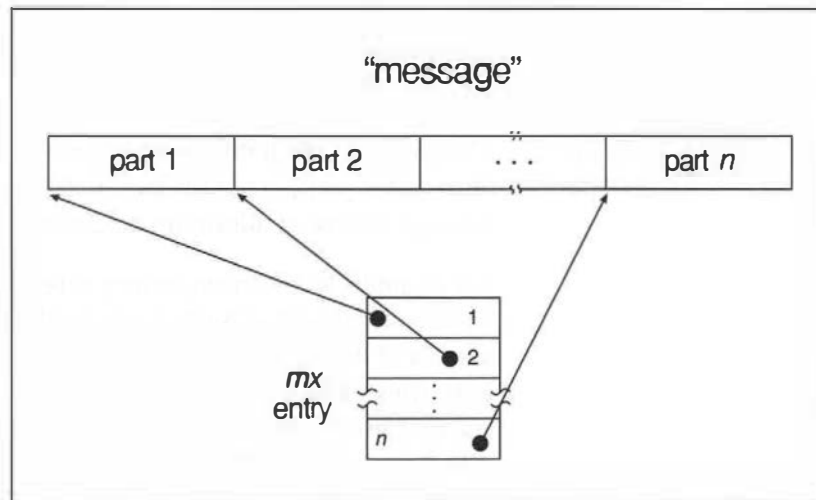
For example, an I/O manager may accept messages of data to be written that consist of a fixed-size header followed by a variable amount of data. The header contains the byte count of the data (0 to 64K bytes). The I/O manager may elect to receive only the header and then use the *Readmsg()* C function to read the variable-length data directly into an appropriate output buffer. If the sent data exceeds the size of the I/O manager's buffer, the manager may issue several *Readmsg()* requests over time to transfer the data as space becomes available. Likewise, the *Writemsg()* C function can be used to collect data over time and copy it back to the sender's reply buffer as it becomes available, thus reducing the I/O manager's internal buffer requirements.

Multi-part messages

Up to now, messages have been discussed as single packets of bytes. However, messages often consist of two or more discrete components. For example, a message may have a fixed-length header followed by a variable amount of data. So that its components will be efficiently sent or received without being copied into a temporary work buffer, a multi-part message can be constructed from two or more separate message buffers. This facility helps QNX I/O managers, such as **Dev** and **Fsys**, achieve their high performance.

The following functions are available to handle multi-part messages:

Creceivemx()
Readmsgmx()
Receivemx()
Replymx()
Sendmx()
Writemsgmx()



Multi-part messages can be specified with an `mx` control structure. The kernel assembles these into a single data stream.

Reserved message codes

Although you aren't required to do so, Quantum begins all of its messages with a 16-bit word called a *message code*. Note that Quantum's system processes use message codes in the following ranges:

0X0000 to 0X00FF	Process Manager messages
0X0100 to 0X01FF	I/O messages (common to all I/O servers)
0X0200 to 0X02FF	Filesystem Manager messages
0X0300 to 0X03FF	Device Manager messages
0X0400 to 0X04FF	Network Manager messages
0X0500 to 0X0FFF	Reserved for future Quantum system processes

IPC via proxies

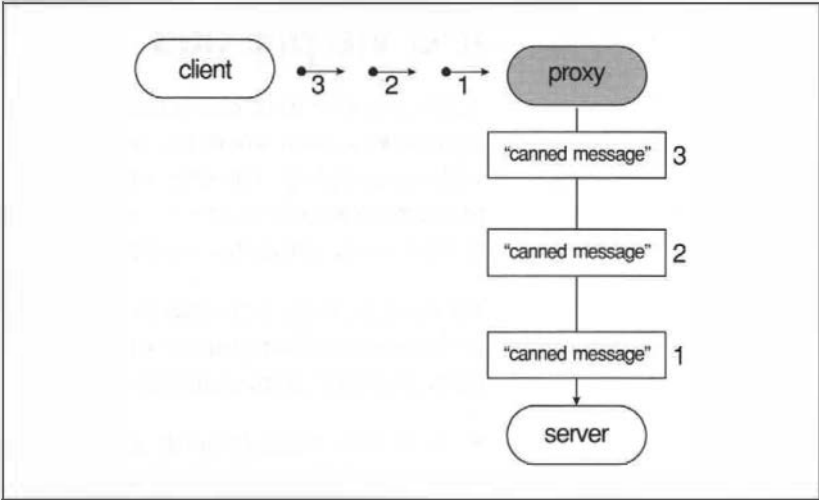
A *proxy* is a form of non-blocking message especially suited for event notification where the sending process doesn't need to interact with the recipient. The only function of a proxy is to send a fixed message to a specific process that owns the proxy. Like messages, proxies work across the network.

By using a proxy, a process or an interrupt handler can send a message to another process without blocking or having to wait for a reply. Here are some examples of when proxies are used:

- A process wants to notify another process that an event has occurred, but can't afford to become SEND-blocked until the recipient issues a *Receive()* and a *Reply()*.
- A process wants to send data to another process, but needs neither a reply nor any other acknowledgment that the recipient has received the message.
- An interrupt handler wants to tell a process that some data is available for processing.

Proxies are created with the *qnx_proxy_attach()* C function. Any other process or any interrupt handler that knows the identification of the proxy can then cause the proxy to deliver its predefined message by using the *Trigger()* C function. The Kernel handles the *Trigger()* request.

A proxy can be triggered more than once—it sends a message for each time it's triggered. A proxy process can queue up to 65,535 messages for delivery.



A client process triggers a proxy three times, which causes the server to receive three "canned" messages from the proxy.

IPC via signals

Signals are a traditional method of asynchronous communication that have been available for many years in a variety of operating systems.

QNX supports a rich set of POSIX-compliant signals, in addition to QNX-specific signals as well as signals historically found on some UNIX systems.

Generating signals

A signal is considered to have been delivered to a process when the process-defined action for that signal is taken. A process can set a signal on itself.

If you want to:	use the:
generate a signal from the shell	kill or slay utilities
generate a signal from within a process	<i>kill()</i> or <i>raise()</i> C functions

Receiving signals

A process can receive a signal in one of three ways, depending on how it has defined its signal-handling environment:

- If the process has taken no special action to handle signals, the default action for the signal is taken—usually, this default action is to terminate the process.
- The process can ignore the signal. If a process ignores a signal, there's no effect on the process when the signal is delivered (note that the SIGCONT, SIGKILL, and SIGSTOP signals can't be ignored under normal circumstances).
- The process can provide a *signal handler* for the signal—a signal handler is a function in the process that is invoked when the signal is delivered. When a process contains a signal handler for a signal, it is said to be able to “catch” the signal. Any process that catches a signal is, in effect, receiving a form of software interrupt. No data is transferred with the signal.

Between the time that a signal is generated and the time that it's delivered, the signal is said to be pending. Several distinct signals can be pending for a process at a given time. Signals are delivered to a process when the process is made ready to run by the Kernel's scheduler. A process should make no assumptions about the order in which pending signals are delivered.

Summary of signals

The following table lists each signal, shows whether the signal may be caught or ignored, and gives the default action taken for the signal.

Signal	Catch or ignore	Default action	Description
standard signals			
SIGABRT	yes	Terminate the process	Abnormal termination signal such as issued by the <i>abort()</i> function
SIGALRM	yes	"	Timeout signal such as issued by the <i>alarm()</i> function
SIGFPE**	yes	"	Erroneous arithmetic operation (integer or floating point), such as division by zero or an operation resulting in overflow
SIGHUP	yes	"	Death of session leader, or hangup detected on controlling terminal
SIGILL**	yes	"	Detection of an invalid hardware instruction
SIGINT	yes	"	Interactive attention signal (<Break>)
SIGKILL*	no	"	Termination signal; should be used only for emergency situations
SIGPIPE	yes	"	Attempt to write on a pipe with no readers
SIGQUIT	yes	"	Interactive termination signal
SIGSEGV**	yes	"	Detection of an invalid memory reference
SIGTERM	yes	"	Termination signal
SIGUSR1	yes	"	Reserved as application-defined signal 1
SIGUSR2	yes	"	Reserved as application-defined signal 2
job control signals			
SIGCHLD	yes	Ignore the signal	Child process terminated
SIGCONT	no	Continue a HELD process; ignore the signal if the process isn't HELD	Continue if HELD

Signal	Catch or ignore	Default action	Description
SIGSTOP*	no	Hold the process	HOLD process signal
SIGTSTP*	no	Ignore the signal	Not supported by QNX
SIGTTIN	no	"	"
SIGTTOU	no	"	"

QNX-specific signals

SIGBUS**	yes	Terminate the process	Indicates a memory parity error (QNX-specific interpretation)
SIGDEV	yes	"	Generated when a significant and requested event occurs in the Device Manager
SIGPWR	yes	"	Soft boot requested via <Ctrl>-<Alt>-<Shift>- or shutdown utility

historical UNIX signals

SIGIOT†	yes	Terminate the process	IOT instruction
SIGSYS†	yes	"	Bad argument to system call
SIGWINCH†	yes	"	Window change
SIGURG†	yes	"	Urgent condition present on socket
SIGPOLL†	yes	"	Pollable event occurred
SIGEMT†	yes	"	EMT instruction (emulator trap)
SIGTRAP	yes	"	Unsupported software interrupt

* A server may protect itself against this signal via the *qnx_pflags()* C function. The server must have superuser privileges to do so.

** If a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.

† This signal is defined for historical compatibility with some UNIX systems; it is *not* generated by any QNX component.

Defining signal processing

To define the type of handling you want for each signal, you use the ANSI C *signal()* function or the POSIX *sigaction()* function.

The *sigaction()* function gives you greater control over the signal handling environment.

You may change the type of handling for a signal at any time. If you set the signal handling for a function to ignore the signal, any pending signals of that type will be immediately discarded.

Catching signals

Some special considerations apply to processes that catch signals with a signal handling function.

The signal handling function is similar to a software interrupt. It is executed asynchronously to the rest of the process. Therefore, it's possible for a signal handler to be entered while any function in the program was running (including library functions).

If your process doesn't return from the signal handler, it can use either the *siglongjmp()* or the *longjmp()* C function, but *siglongjmp()* is preferred. With *longjmp()*, the signal remains blocked.

Safe functions for signal handlers

The POSIX and ANSI C library functions listed below are specified as being safe to use within signal handlers. You shouldn't try to use any other library functions, since the results of doing so are unspecified and unpredictable. Nor should you try to use any user functions in your program unless they are re-entrant.

<i>_exit()</i>	<i>getegid()</i>	<i>rmdir()</i>	<i>tcdrain()</i>
<i>access()</i>	<i>geteuid()</i>	<i>setgid()</i>	<i>tcflow()</i>
<i>alarm()</i>	<i>getgid()</i>	<i>setpgid()</i>	<i>tcflush()</i>
<i>cfgetispeed()</i>	<i>getgroups()</i>	<i>setsid()</i>	<i>tcgetattr()</i>
<i>cfgetospeed()</i>	<i>getpgrp()</i>	<i>setuid()</i>	<i>tcgetpgrp()</i>
<i>cfsetispeed()</i>	<i>getpid()</i>	<i>sigaction()</i>	<i>tscendbreak()</i>
<i>cfsetospeed()</i>	<i>getppid()</i>	<i>sigaddset()</i>	<i>tcsetattr()</i>
<i>chdir()</i>	<i>getuid()</i>	<i>sigdelset()</i>	<i>tcsetgrp()</i>
<i>chmod()</i>	<i>kill()</i>	<i>sigemptyset()</i>	<i>time()</i>
<i>chown()</i>	<i>link()</i>	<i>sigfillset()</i>	<i>times()</i>
<i>close()</i>	<i>lseek()</i>	<i>sigismember()</i>	<i>umask()</i>
<i>creat()</i>	<i>mkdir()</i>	<i>signal()</i>	<i>uname()</i>
<i>dup2()</i>	<i>mkfifo()</i>	<i>sigpending()</i>	<i>unlink()</i>
<i>dup()</i>	<i>open()</i>	<i>sigprocmask()</i>	<i>ustat()</i>
<i>execle()</i>	<i>pathconf()</i>	<i>sigsuspend()</i>	<i>utime()</i>
<i>execve()</i>	<i>pause()</i>	<i>sleep()</i>	<i>wait()</i>
<i>fcntl()</i>	<i>pipe()</i>	<i>stat()</i>	<i>waitpid()</i>
<i>fork()</i>	<i>read()</i>	<i>sysconf()</i>	<i>write()</i>
<i>fstat()</i>	<i>rename()</i>		

Blocking signals

Sometimes you may want to temporarily prevent a signal from being delivered, without changing the method of how the signal is handled when it is delivered. QNX provides a set of functions that let you block delivery of signals. A signal that is blocked remains pending; once unblocked, it is delivered to your program.

While your process is executing a signal handler for a particular signal, QNX automatically blocks that signal. This means that you don't have to worry about setting up nested invocations of your handler. Each invocation of your signal handler is an atomic operation with respect to the delivery of further signals of that type. If your process returns normally from the handler, the signal is automatically unblocked.



Some UNIX systems have a flawed implementation of signal handlers in that they reset the signal to the default action rather than block the signal. As a result, some UNIX applications call the *signal()* function within the signal handler to re-arm the handler. This has two windows of failure. First, if another signal arrives while your program is in the handler but before *signal()* is called, your program may be killed. Second, if a signal arrives just after the call to *signal()* in the handler, you might enter your handler recursively. QNX supports signal blocking and therefore suffers from neither of these problems. You **don't** need to call *signal()* within your handler. If you leave your handler via a long jump, you should use the *siglongjmp()* function.

Signals and messages

There's an important interaction between signals and messages. If your process is SEND-blocked or RECEIVE-blocked when a signal is generated—and you have a signal handler—the following actions occur:

- 1) the process is unblocked
- 2) signal handling processing takes place
- 3) the *Send()* or *Receive()* returns with an error

If your process was SEND-blocked at the time, this doesn't represent a problem, because the recipient wouldn't have received a message. But if your process was REPLY-blocked, you won't know whether the sent message had been handled or not, and therefore won't know whether to retry the *Send()*.

It's possible for a process acting as a server (i.e. it is receiving messages) to ask that it be notified when a client process is signaled while in the REPLY-blocked state. In this case, the client process is made SIGNAL-blocked with a pending signal and the server process receives a special message describing the type of signal. The server process can then decide to do either of the following:

- complete the original request normally—the sender is assured that the message was handled properly
- release any resources tied up and return an error indicating that the process was unblocked by a signal—the sender receives a clear error indication

When the server replies to a process that was SIGNAL-blocked, the signal will take effect immediately after the sender's *Send()* returns.

IPC across the network

Virtual circuits

A QNX application can talk to a process on another computer on the network just as if it were talking to another process on the same machine. As a matter of fact, from the application's perspective, there's no difference between a local and remote resource.

This remarkable degree of transparency is made possible by *virtual circuits* (VCs), which are paths the Network Manager provides to transmit messages, proxies, and signals across the network.

VCs contribute to efficient overall use of resources in a QNX network for several reasons:

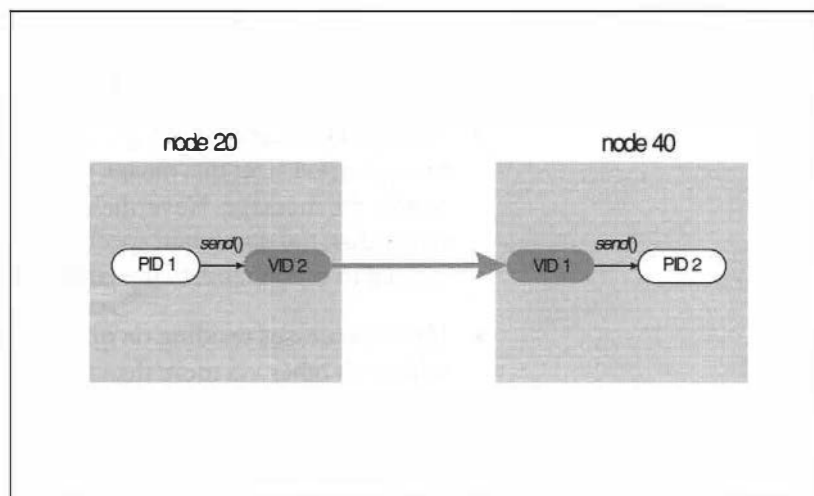
- When a VC is created, it's given the ability to handle messages up to a specified size; this means you can preallocate resources to handle the message. Nevertheless, if you need to send a message larger than the maximum specified size, the VC is automatically resized to accommodate the larger message.
- If two processes residing on different nodes are communicating with each other via more than one VC, the VCs are shared---only one *real* virtual circuit exists between the processes. This situation occurs commonly when a process accesses several files on a remote filesystem.
- If a process attaches to an existing shared VC and it requests a buffer size larger than that currently in use, the buffer size is automatically increased.
- When a process terminates, its associated VCs are automatically released.

Virtual processes

A sending process is responsible for setting up the VC between itself and the process it wants to communicate with. To do so, the sending process usually issues a *qnx_vc_attach()* function call. In addition to creating a VC, this call also creates a virtual process ID, or VID, at each end of the circuit. To the process at either end of the virtual

circuit, the VID on its end appears to have the process ID of the remote process it wants to communicate with. Processes communicate with each other via these VIDs.

For example, in the following illustration, a virtual circuit connects PID 1 to PID 2. On node 20—where PID 1 resides—a VID represents PID 2. On node 40—where PID 2 resides—a VID represents PID 1. Both PID 1 and PID 2 can refer to the VID on their node as if it were any other local process, sending messages, receiving messages, raising signals, waiting, etc. So, for example, PID 1 can send a message to the VID on its end, and this VID will relay the message across the network to the VID representing PID 1 on the other end. This VID will then route the message to PID 2.



Network communications is handled with virtual circuits. When PID 1 sends to VID 2, the send request is relayed across the virtual circuit causing VID 1 to send to PID 2.

Each VID maintains a connection that contains the following information:

- local *pid*
- remote *pid*
- remote *nid* (node ID)
- remote *vid*

You probably won't come into direct contact with VCs very often. For example, when an application wants to access an I/O resource across the network, a VC is created by the *open()* library function on the application's behalf. The application has no direct part in the creation or use of the VC. Again, when an application establishes the location of a server with the *qnx_name_locate()* C function, a VC is automatically created on behalf of the application. To the application, the VC simply appears to be a PID.

For more information on *qnx_name_locate()*, see the discussion of process symbolic names in Chapter 3.

Terminating virtual circuits

A process might become unable to communicate over an established VC for various reasons, including:

- The computer it was running on was powered down.
- The network cable to the computer was disconnected.
- The remote process it was communicating with was terminated.

Any of these conditions can prevent messages from being transmitted over a VC. It's necessary to detect these situations so that applications can take remedial action or terminate themselves gracefully. If this isn't done, valuable resources can be unnecessarily tied up.

The Process Manager on each node checks the integrity of the VCs on its node. It does this as follows:

- 1) Each time a successful transmission takes place on a VC, a time stamp associated with the VC is updated to indicate the time of last activity.
- 2) At installation-defined intervals, the Process Manager looks at each VC. If there's been no activity on a circuit, the Process Manager sends a network integrity packet to the Process Manager on the node at the other end of the circuit.
- 3) If no response comes back, or if a problem is indicated, the VC is flagged as having a problem. An installation-defined number of attempts are then made to re-establish contact.

- 4) If the attempts fail, the VC is dismantled; any process blocked on the VC is made READY. (The process sees a failure return code from the communication primitive on the VC.)

To control parameters related to this integrity check, you use the `netpoll` utility.

Process scheduling

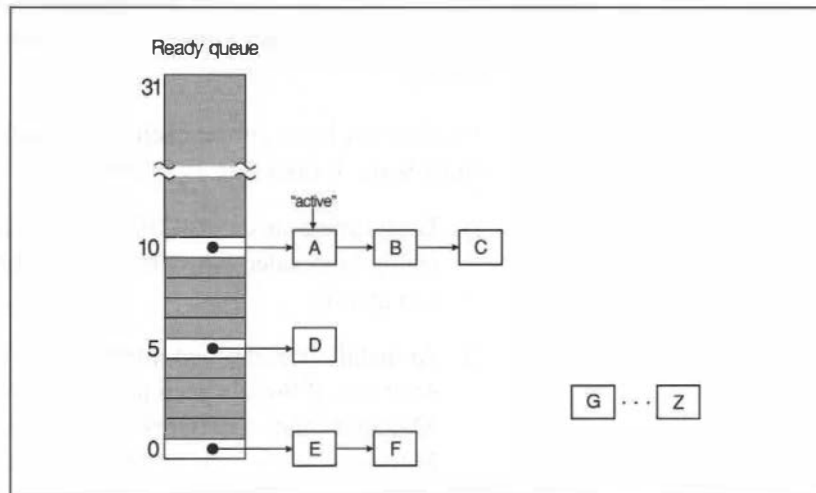
**When
scheduling
decisions are
made**

The Kernel's scheduler makes scheduling decisions when:

- a process becomes unblocked
- the timeslice for a running process expires
- a running process is preempted

Process priorities

In QNX, every process is assigned a priority. The scheduler selects the next process to run by looking at the priority assigned to every process that is READY (a READY process is one capable of using the CPU). The process with the highest priority is selected to run.



The ready queue for six processes (A-F) which are READY. All other processes (G-Z) are BLOCKED. Process A is currently running. Processes A, B, and C are at the highest priority, so will share the processor based on the running process's scheduling algorithm.

The priorities assigned to processes range from 0 (the lowest) to 31 (the highest). The default priority for a new process is inherited from its parent; this is normally set to 10 for applications started by the Shell.

If you want to:	use the:
determine the priority of a process	<i>getprio()</i> C function
set the priority of a process	<i>setprio()</i> C function

Scheduling methods

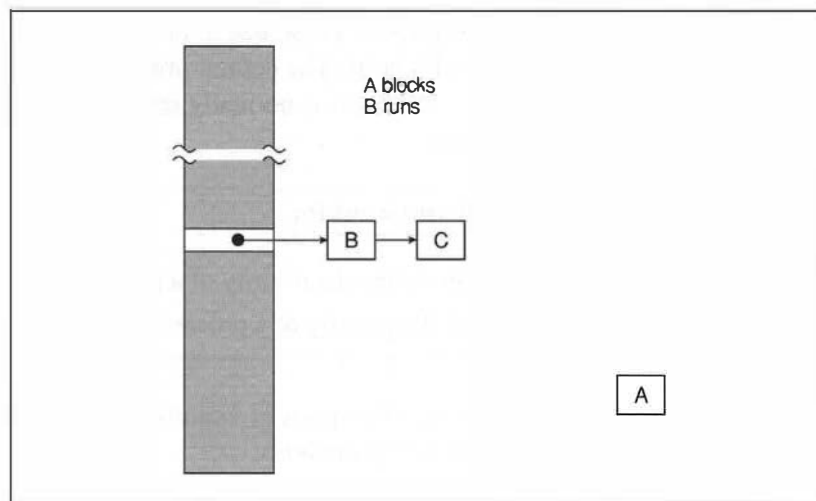
To meet the needs of various applications, QNX provides three scheduling methods:

- FIFO scheduling
- round-robin scheduling
- adaptive scheduling

Each process on the system may run using any one of these methods. They are effective on a per-process basis, not on a global basis for all processes on a node.

Remember that these scheduling methods apply only when two or more processes that share the same priority are READY (i.e. the processes are directly competing with each other). If a higher-priority process becomes READY, it immediately preempts all lower-priority processes.

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



Process A blocks, Process B runs.

The scheduling method of a process is inherited from its parent process—the method can be changed, however.

If you want to:	use the:
determine the scheduling method for a process	<code>getscheduler()</code> C function
set the scheduling method for a process	<code>setscheduler()</code> C function

FIFO scheduling

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

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

Two process that run at the same priority can use FIFO scheduling to ensure mutual exclusion to a shared resource. Neither process will be preempted by the other while it is executing. For example, if they shared a memory segment, each of the two processes could update the segment without resorting to some form of semaphoring.

Round-robin scheduling

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

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

A timeslice is the unit of time assigned to every process. Once it consumes its timeslice, a process is preempted and the next READY process at the same priority level is given control. A timeslice is 100 milliseconds.



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

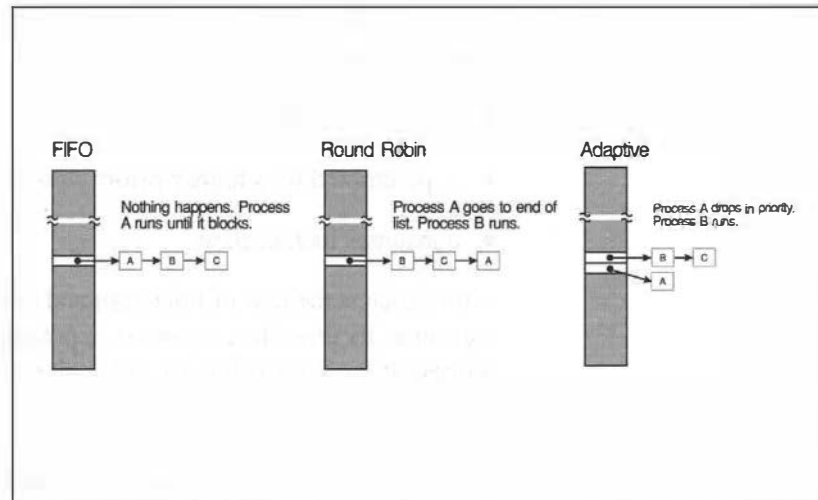
Adaptive scheduling

In adaptive scheduling, a process behaves as follows:

- If the process consumes its timeslice (i.e. it doesn't block), its priority is reduced by 1 if another process at the same priority is READY. This is known as *priority decay*.
- If the process has decayed and remains unscheduled for one second, its priority is boosted by 1 (a process is never boosted above its original priority).
- If the process blocks, it immediately reverts to its original priority.

You can use adaptive scheduling in environments where potentially compute-intensive background processes are sharing the computer with interactive users. You should find that adaptive scheduling gives the compute-intensive processes sufficient access to the CPU, yet retains fast interactive response for other processes.

Adaptive scheduling is the default scheduling method for programs created by the Shell.



Process A continues to run until a timeslice occurs, at which point the scheduling algorithm determines what happens next.

Client-driven priority

In QNX, most transactions between processes follow a *client/server* model. *Servers* provide some form of service and *clients* send messages to these servers to request service. In general, servers are more trusted and vital than clients.

Clients usually outnumber servers. As a result, it is common to run a server at a priority that exceeds the priorities of all its clients. The scheduling method may be any of the three previously described, but round-robin is probably the most common.

If a low-priority client sends a message to the server, then its request will by default be handled at the higher priority of the server. This has, in an indirect way, boosted the priority of the client since it is the client's request that is causing the server to run. As long as the server runs for a short period of time to satisfy the request, this usually isn't a concern. If the server runs for a more extended period, then a low-priority client may adversely affect other processes at priorities higher than the client but lower than the server.

To solve this dilemma, a server may elect to have its priority driven by the priority of the clients that send it messages. When the server receives a message, its priority will be set to that of the client. Note that only its priority is changed—its scheduling method stays the

same. If another message arrives while the server is running, the server's priority will be boosted if the new client's priority is greater than the server's. In effect, the new client "turbocharges" the server to its priority, allowing it to finish the current request so it can handle the new client's request. If this weren't done, the new client would in effect have its priority lowered as it blocked on a lower-priority server.

If you select client-driven priorities for your server, you should also request that messages be delivered in priority order (as opposed to time order).

To enable client-driven priority, you use the `qnx_pflags()` C function as follows:

```
qnx_pflags(~0, _PPF_PRIORITY_FLOAT
            | _PPF_PRIORITY_REC, 0, 0);
```

A word about realtime performance

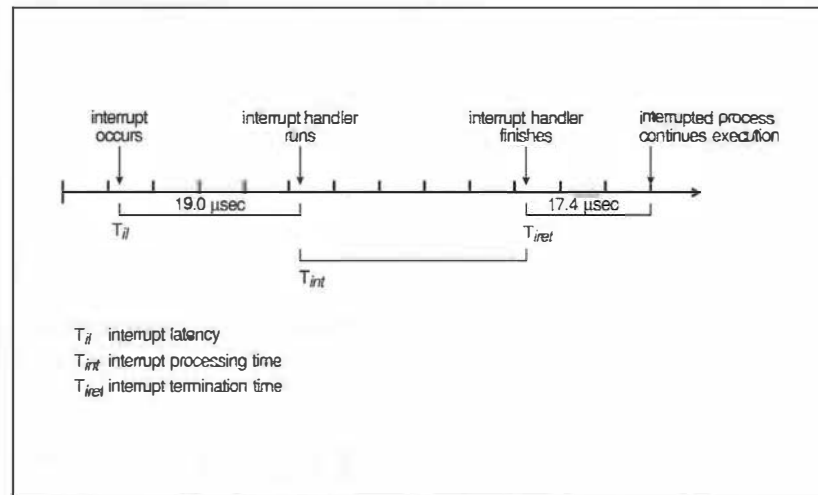
No matter how much we wish it were so, computers are not infinitely fast. In a realtime system, it's absolutely crucial that CPU cycles aren't unnecessarily spent. It's also crucial that you minimize the time it takes from the occurrence of an external event to the actual execution of code within the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

Several forms of latency are encountered in a QNX system.

Interrupt latency

Interrupt latency is the time from the reception of a hardware interrupt until the first instruction of a software interrupt handler is executed. QNX leaves interrupts fully enabled almost all of 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 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 a proxy to be triggered.



Interrupt handler simply terminates. Times are for a 20Mhz 386 processor in protected mode.

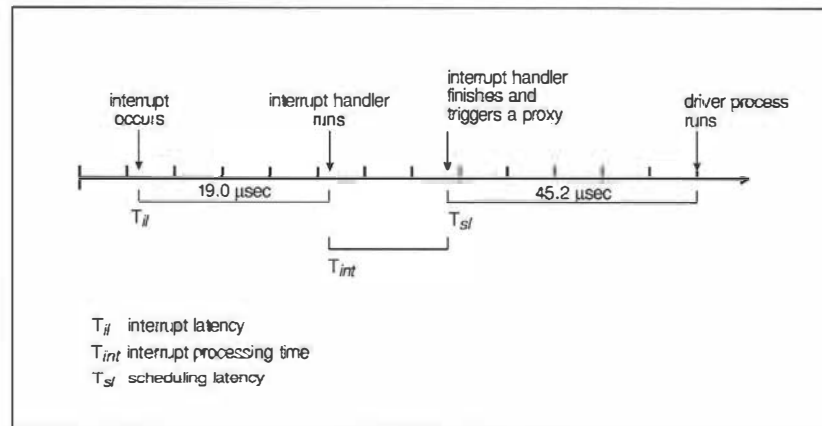
The interrupt latency (T_{ii}) in the above diagram represents the *minimum* latency—that which occurs when interrupts were fully enabled at the time the interrupt occurred. Worst-case interrupt latency will be this time plus the longest time in which QNX, or the running QNX process, disables CPU interrupts.

Scheduling latency

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

Scheduling latency is the time between the termination of an interrupt handler and the execution of the first instruction of a driver process. This usually means the time it takes to save the context of

the currently executing process and restore the context of the required driver process. Although larger than interrupt latency, this time is also kept small in a QNX system.



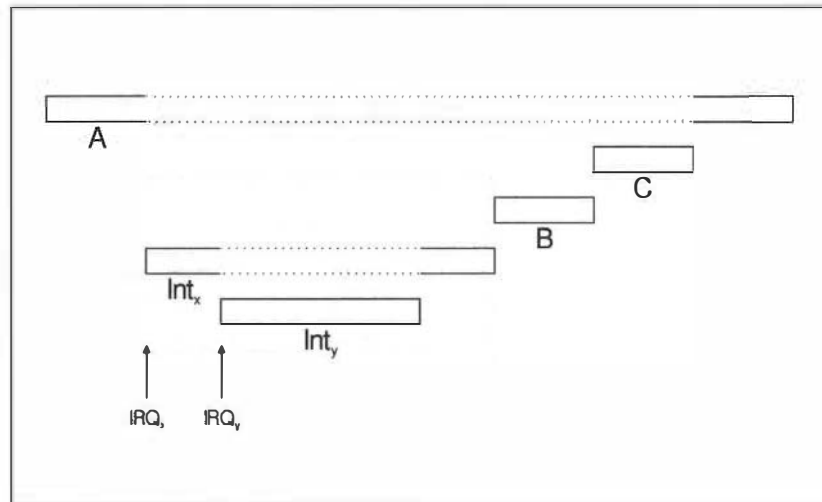
Interrupt handler terminates and triggers a proxy. Times are for a 20Mhz 386 processor in protected mode.

It's important to note that *most* interrupts terminate without triggering a proxy. In a large number of cases, the interrupt handler can take care of all hardware-related issues. Triggering a proxy to kick a higher-level *driver* process 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 process (**Dev**) only when the output buffer is finally empty.

Stacked interrupts

Since microcomputer architectures allow hardware interrupts to be given priorities, higher-priority interrupts can preempt a lower-priority interrupt.

This mechanism is fully supported in QNX. The previous scenarios describe the simplest—and most common—situation where only one interrupt occurs. Substantially similar timing is true for the highest priority interrupt. Worst-case timing considerations for lower-priority interrupts must take into account the time for all higher-priority interrupts to be processed, since in QNX, a higher-priority interrupt will preempt a lower-priority interrupt.



Process 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 triggers a proxy causing process B to run, and Int_x triggers a proxy causing process C to run.

Chapter 3

The Process Manager

In this Chapter...

Introduction

The life cycle of a process

Process states

Process symbolic names

Timers

Interrupt handlers

Introduction

Process Manager responsibilities

The Process Manager works closely with the Kernel to provide essential operating system services. Although it shares the same address space as the Kernel, and is the only process to do so, the Process Manager runs as a true process. As such, it is scheduled to run by the Kernel like all other processes and it uses the Kernel's message-passing primitives to communicate with other processes in the system.

The Process Manager is responsible for creating new processes in the system and managing the most fundamental resources associated with a process. These services are all provided via messages. For example, if a running process wants to create a new process, it does so by sending a message containing the details of the new process to be created. Note that since messages are network-wide, you can easily create a process on another node by sending the process creation message to the Process Manager on that node.

Process creation primitives

QNX supports three process creation primitives:

fork()
exec()
spawn()

Both *fork()* and *exec()* are defined by POSIX, while the implementation of *spawn()* is unique to QNX.

fork() The *fork()* primitive creates a new process that is an exact image of the calling process. The new process shares the same code as the calling process and inherits a copy of all of the calling process's data.

exec() The *exec()* primitive replaces the calling process image with a new process image. There's no return from a successful *exec()*, because the calling process image is overlaid by the new process image. It's common practice in POSIX systems to create a new process—without removing the calling process—by first calling *fork()*, and then having the child of the *fork()* call *exec()*.

spawn() The *spawn()* primitive creates a new process as a child of the calling process. It can avoid the need to *fork()* and *exec()*, resulting in a faster and more efficient means for creating new processes. Unlike

fork() and *exec()*, which by their very nature operate on the same node as the calling process, the *spawn()* primitive can create processes *on any node in the network*.

Process inheritance

When a process is created by one of the three primitives described above, it inherits much of its environment from its parent. This is summarized in the following table:

Item inherited	<i>fork()</i>	<i>exec()</i>	<i>spawn()</i>
process ID	no	yes	no
open files	yes	optional*	optional
file locks	no	yes	no
pending signals	no	yes	no
signal mask	yes	optional	optional
ignored signals	yes	optional	optional
signal handlers	yes	no	no
environment variables	yes	optional	optional
session ID	yes	yes	optional
process group	yes	yes	optional
real UID, GID	yes	yes	yes
effective UID, GID	yes	optional	optional
current working directory	yes	optional	optional
file creation mask	yes	yes	yes
priority	yes	optional	optional
scheduler policy	yes	optional	optional
virtual circuits	no	no	no
symbolic names	no	no	no
realtime timers	no	no	no

* optional: the caller may select either yes or no, as needed.

The life cycle of a process

A process goes through four phases:

- 1) Creation
- 2) Loading
- 3) Execution
- 4) Termination

Creation

Creating a process consists of allocating a process ID for the new process and setting up the information that defines the environment of the new process. Most of this information is inherited from the parent of the new process (see the section on “Process inheritance”).

Loading

The loading of process images is done by a *loader thread*. The loader code resides in the Process Manager, but the thread runs under the process ID of the new process. This lets the Process Manager handle other requests while loading programs.

Execution

Once the program code has been loaded, the process is ready for execution; it begins to compete with other processes for CPU resources. Note that all processes run concurrently with their parents. In addition, the death of a parent process does *not* automatically cause the death of its child processes.

Termination

A process is terminated in either of two ways:

- a signal whose defined action is to cause process termination is delivered to the process
- the process invokes the *exit()* C function, either explicitly or by default action when returning from *main()*

Termination involves two stages:

- 1) A *termination thread* in the Process Manager is run. This “trusted” code is in the Process Manager but the thread runs with the process ID of the terminating process. This thread closes all open file descriptors and releases the following:

- any virtual circuits held by the process
- all memory allocated to the process
- any symbolic names
- any major device numbers (I/O managers only)
- any interrupt handlers
- any proxies
- any timers

- 2) After the termination thread is run, notification of process termination is sent to the parent process (this phase runs inside the Process Manager).

If the parent process hasn't issued a *wait()* or *waitpid()* call, the child process becomes a "zombie" process and won't terminate until the parent process issues a *wait()* or terminates.

(If you don't want a process to *wait* on the death of children, you should either set the `_SPAWN_NOZOMBIE` flag with *qnx_spawn()* or *qnx_spawn_options()* or set the action for `SIGCHLD` to be `SIG_IGN` via *signal()*. This way, children won't become zombies when they die.)

A parent process can *wait* on the death of a child spawned on a remote node. If the parent of a zombie process terminates, the zombie is released.

If a process is terminated by the delivery of a termination signal and the **dumper** utility is running, a memory-image dump is generated. You can examine this dump with the symbolic debugger.

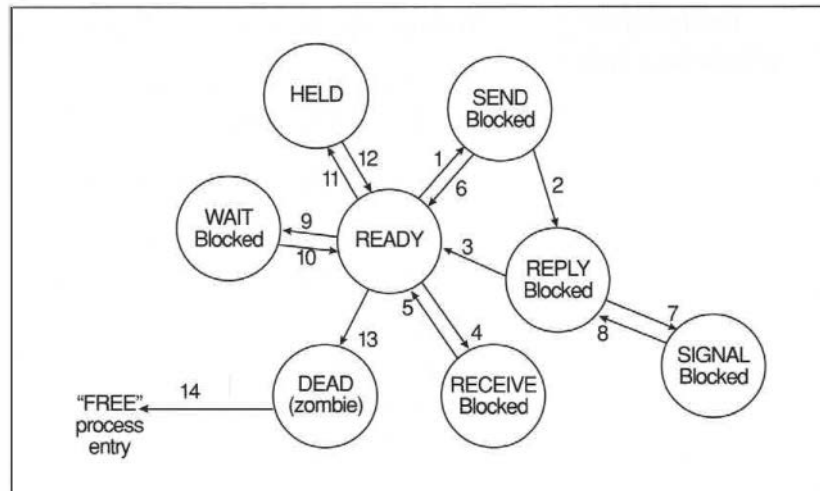
Process states

A process is always in one of the following states:

- **READY**—the process is capable of using the CPU (i.e. it isn't waiting for any event to occur).
- **BLOCKED**—the process is in one of the following blocked states:
 - SEND-blocked
 - RECEIVE-blocked
 - REPLY-blocked
 - SIGNAL-blocked
- **HELD**—the process has received a SIGSTOP signal. Until it's removed from the **HELD** state, a process isn't eligible to use the CPU; the only way to remove it from the **HELD** state is to either deliver a SIGCONT signal or terminate the process via a signal.
- **WAIT-blocked**—the process has issued a *wait()* or *waitpid()* call to wait for status from one or more of its child processes.
- **DEAD**—the process has terminated but is unable to send its exit status to its parent because the parent hasn't issued a *wait()* or *waitpid()*. A DEAD process has a state, but the memory it once occupied has been released. A DEAD process is also known as a *zombie* process.



For more information on blocked states, see Chapter 2, The Microkernel.



Possible process states in a QNX system. The transactions are as follows:

- 1 Process sends message.
- 2 Target process receives message.
- 3 Target process replies.
- 4 Process waits for message.
- 5 Process receives message.
- 6 Signal unblocks process.
- 7 Signal attempts to unblock process;
target has requested message signal catching.
- 8 Target process receives signal message.
- 9 Process waits on death of child.
- 10 Child dies or signal unblocks process.
- 11 SIGSTOP set on process.
- 12 SIGCONT set on process.
- 13 Process dies.
- 14 Parent waits on death, terminates itself or has already terminated.

Determining process states

To determine the state of an individual process:

from within the Shell, use the:	ps and sin utilities
from within applications, use the:	qnx_psinfo() function

To determine the state of the operating system as a whole:

from within the Shell, use the:	sin utility
from within applications, use the:	qnx_osinfo() function

The **ps** utility is defined by POSIX; its use in command scripts *may* be portable. The **sin** utility, on the other hand, is unique to QNX; it gives you helpful QNX-specific information that you can't get from the **ps** utility.

Process symbolic names

QNX encourages the development of applications that are split up into cooperating processes. An application that exists as a team of cooperating processes exhibits greater concurrency and can be network-distributed for greater performance.

Splitting up applications into cooperating processes requires special considerations, however. If cooperating processes are to communicate reliably with each other, they must be able to determine each other's process ID. For example, let's say you have a database server process that provides services to an arbitrary number of clients. The clients start and stop at any time, but the server always remains available. How do client processes discover the process ID of the database server so they can send messages to it?

In QNX, the solution is to let processes give themselves a symbolic name. In the context of a single node, processes can register this name with the Process Manager on the node where they're running. Other processes can then ask the Process Manager for the process ID associated with that name.

The situation becomes more complex in a network environment where a server may need to service clients from several nodes across a network. QNX accordingly provides the ability to support both

global names and local names. Global names are known across the network, whereas local names are known only on the node where they are registered. Global names start with a slash (/). For example:

qnx/fsys	local name
company/xyz	local name
/company/xyz	global name



We recommend that you prefix all your names with your company name to reduce name conflicts among vendors.

In order for global names to be used, a process known as a *process name locator* (i.e. the **nameloc** utility) must be running on at least one node of a network. This process maintains a record of all global names that have been registered.

Up to ten process name locators may be active on a network at a particular time. Each maintains an identical copy of all active global names. This redundancy ensures that a network can continue to function properly even if one or more nodes supporting process name location fail simultaneously.

To attach a name, a *server* process uses the *qnx_name_attach()* C function. To locate a process by name, a *client* process uses the *qnx_name_locate()* C function.

Timers

Time management

In QNX, time management is based on a system timer maintained by the operating system. The timer contains the current Coordinated Universal Time (UTC) relative to 0 hours, 0 minutes, 0 seconds, January 1, 1970. To establish local time, time management functions use the **TZ** environment variable (which is described in the *User's Guide*).

Simple timing facilities

Shell scripts and processes can pause for a specific number of seconds, using a simple timing facility. For shell scripts, this facility is provided by the **sleep** utility; for processes, it's provided by the *sleep()* C function. You can also use the *delay()* function, which takes a time interval specified in milliseconds.

Advanced timing facilities

Creating timers

A process can also create timers, arm them with a time interval, and remove timers. These advanced timing facilities are based on the POSIX Std 1003.4/Draft 9 specification.

A process can create one or more timers. These timers can be any mix of supported timer types, subject to a configurable limit on the total number of timers supported by the operating system (see **Proc** in the *Utilities Reference*).

To create timers, you use the *mktimer()* C function. This function lets you specify the type of event-reporting mechanism to be used:

- **sleep until completion**—the process will sleep from the time it issues the timer arming call until the timer interval expires
- **notify with proxy**—a proxy will be used to notify the process that the time interval has expired
- **notify with signal**—a user-specified signal will be delivered to the process when the time interval expires

Arming timers

You can arm timers with the following time intervals:

- **absolute**—the time relative to 0 hours, 0 minutes, 0 seconds, January 1, 1970
- **relative**—the time relative to the current clock value

You can also have a timer go off repeatedly at a specified interval. For example, let's say you have a timer armed to go off at 9 am tomorrow morning. You can specify that it should also go off every five minutes thereafter.

You can also set a new time interval on an existing timer. The effect of the new time interval depends on the interval type:

- for an absolute timer, the new interval *replaces* the current time interval
- for a relative timer, the new interval is *added* to any remaining time interval

	To arm timers with:	use the:
	an absolute time interval	<i>abstimer()</i> C function
	a relative time interval	<i>reltimer()</i> C function
Removing timers	To remove timers, you use the <i>rmtimer()</i> C function. A timer can also remove itself when its time interval expires, provided these conditions are met: - the <code>_TNOTIFY_SLEEP</code> option was included in the <i>rmtimer()</i> function call - the timer wasn't armed to go off repeatedly	
Setting timer resolution	You can set the resolution of timers via the <code>ticksize</code> utility or the <i>qnx_timerperiod()</i> C function. You can adjust the resolution from 500 μ sec to 50 milliseconds.	
Reading timers	To inspect the interval outstanding on a timer, or to check if the timer has been removed, you use the <i>gettimer()</i> C function.	

Interrupt handlers

Interrupt handlers service the computer's hardware interrupt system—they react to hardware interrupts and manage the low-level transfer of data between the computer and external devices.

Interrupt handlers are physically packaged as part of a standard QNX process (e.g. a driver), but they always run asynchronously to the process they're associated with.

An interrupt handler:

- is entered with a far call, not directly from the interrupt itself (this can be written in C, rather than in assembler)
- runs in the context of the process it is embedded in, so it has access to all the global variables of the process

- runs with interrupts enabled, but is preempted only if a higher-priority interrupt occurs
- shouldn't talk directly to the 8259 interrupt hardware (the operating system takes care of this)

Several processes may attach to the same interrupt (if supported by the hardware). When a physical interrupt occurs, each interrupt handler in turn will be given control. No assumptions should be made about the order in which interrupt handlers sharing an interrupt are invoked.

If you want to:	use the:
establish a hardware interrupt	<i>qnx_hint_attach()</i> C function
remove a hardware interrupt	<i>qnx_hint_detach()</i> C function

Timer interrupt handlers

You can attach an interrupt handler directly to the system timer so that the handler will be invoked on each timer interrupt. To set the period, you can use the `ticksize` utility.

You can also attach to a scaled timer interrupt that will activate every 100 milliseconds, regardless of the *tick size*. These timers offer a lower-level alternative to the POSIX 1003.4 timers.

Chapter 4

I/O Namespace

In this Chapter...

Introduction

Resolving pathnames

File descriptor namespace

Introduction

I/O namespace

The I/O resources of QNX are *not* built into the Kernel. Instead, the I/O services are provided by processes that may be started dynamically while the system is running. Since the QNX filesystem is optional, the pathname space isn't built into the filesystem as it is in most monolithic systems.

Prefixes and regions of authority

With QNX, the pathname space is divided into regions of authority. Any process that wishes to provide file-oriented I/O services will register a prefix with the Process Manager defining the portion of the namespace it wishes to administer (i.e. its region of authority). These prefixes make up a prefix tree that is maintained in memory on each computer running QNX.

Resolving pathnames

I/O manager prefixes

When a process opens a file, the file's pathname is applied against the prefix tree in order to direct the *open()* to the appropriate I/O resource manager. For example, the character Device Manager (**Dev**) usually registers the prefix **/dev**. If a process calls *open()* with **/dev/xxx**, a prefix match of **/dev** will occur and the *open()* will be directed to **Dev** (its owner).

The prefix tree may contain partially overlapping regions of authority. In this case, the longest match wins. For example, suppose we have three prefixes registered:

/	disk-based filesystem (Fsys)
/dev	character device system (Dev)
/dev/hd0	raw disk volume (Fsys)

The Filesystem Manager has registered two prefixes, one for a mounted QNX filesystem (i.e. **/**) and one for a block special file that represents an entire physical hard drive (i.e. **/dev/hd0**). The character Device Manager has registered a single prefix. The following illustrates the longest match rule for pathname resolution.

This pathname:	matches:	and resolves to:
/dev/con1	/dev	Dev
/dev/hd0	/dev/hd0	Fsys
/usr/dtdodge/test	/	Fsys

The prefix tree is managed as a list of prefixes separated by colons as follows:

prefix=pid, unit: prefix=pid, unit: prefix=pid, unit

where *pid* is the process ID of the I/O resource manager, and the *unit* number is a single character used by the manager to select one of possibly several prefixes it owns. In the above example, if **Fsys** were process 3 and **Dev** were process 5, then the system prefix tree might look like this:

/dev/hd0=3, a: /dev=5, a: /=3, e

If you want to:	use the:
display the prefix tree	prefix utility
obtain the prefix tree from within a C program	<i>qnx_prefix_query()</i> function

Network root

QNX supports the concept of a super or network root that lets you apply a pathname against a specific node's prefix tree, which is specified by a pathname that starts with *two* slashes followed by a node number. This also lets you easily access files and devices that aren't in your node's normal pathname space. For example, in a typical QNX network the following paths would map as follows:

/dev/ser1	local serial port
//10/dev/ser1	serial port on node 10
//0/dev/ser1	local serial port
//20/usr/dtdodge/test	file on node 20

Note that //0 always refers to the local node.

Default network root

When a program is executed remotely, you typically want any given pathnames to be resolved within the context of your own node's pathname space. For example, this command:

```
//5 ls /
```

which executes **ls** on node 5, should return the same thing as:

```
ls /
```

which executes **ls** on your node. In both cases, **/** should be resolved through your node's prefix tree, not through the one on node 5. Otherwise, imagine the chaos that would result if node 5's root (**/**) were its local hard disk and your node's root were a hard disk local to your machine—executing remotely would get files from a completely different filesystem!

To resolve pathnames properly, each process has associated with it a *default network root* that defines which node's prefix tree will be used to resolve any pathnames starting with a single slash. When a pathname starting with a single **/** is resolved, the default network root is prepended to it. For example, if a process had a default network root of **//9**, then:

```
/usr/home/luc
```

would be resolved as:

```
//9/usr/home/luc
```

which can be interpreted as “resolve via node 9's prefix tree the pathname **/usr/home/luc**”.

The default network root is inherited by new processes when they're created, and is initialized to your local node when your system starts up. For example, let's say you're running on node 9, sitting in a shell with its default network root set to node 9 (a very typical case). If you were to run the command:

```
ls /
```

the command would inherit the default network root of **//9** and you would get back the equivalent of:

```
ls //9/
```

Likewise, if you were to enter this command:

```
//5 ls /
```

you would be starting the **ls** command on node 5, but it would still inherit a default network root of **//9**, so you would again get back the equivalent of **ls //9/**. In both cases the pathname is resolved according to the same pathname space.

If you want to:	use the:
get your current default network root	<i>qnx_prefix_getroot()</i> C function
set your default network root	<i>qnx_prefix_setroot()</i> C function
run a program with a new default network root	on utility

Passing pathnames between processes

If you have several processes running, they may not all have the same default network root—even if they’re running on the same node. For example, one process may have inherited a default network root from its parent process elsewhere on the network or it may have had its default network root explicitly overridden by its parent process.

When passing a pathname between processes whose network roots may differ (e.g. when submitting a file to a spooler for printing), you should prepend the default network root to the pathname before passing the pathname to the recipient process. If you’re sure the sending process and the recipient process have the same default network root (or if the pathname already has a leading *//node/*), then you may omit this step in the sending process.

Alias prefixes

We have discussed prefixes that map to an I/O resource manager. A second form of prefix, known as an *alias prefix*, is a simple string substitution for a matched prefix. An alias prefix is of the form:

prefix=replacement-string

For example, assume you're running on a machine that doesn't have a local filesystem (so there's no process to administer "/"). However, there's a filesystem on another node (say 10) that you wish to access as "/". You accomplish this using the following alias prefix:

/=/10/

This will cause the leading slash (/) to be mapped into the **//10/** prefix. For example, **/usr/dtdodge/test** will be replaced with the following:

//10/usr/dtdodge/test

This new pathname will again be applied against the prefix tree; this time, however, the prefix tree on node 10 will be used because of the leading **//10** pathname. This will resolve to the Filesystem Manager on node 10, where the *open()* request will be directed. With just a few characters, this alias 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:

/dev=5, a: /=/10/

With this prefix tree, pathnames under **/dev** 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 aliases to create special device names. For example, if a print spooler were running on node 20, you could alias a local printer pathname to it as follows:

/dev/printer=/20/dev/spool

Any request to open `/dev/printer` will be directed across the network to the real spooler. Likewise, if a local floppy drive didn't exist, an alias to a remote floppy on node 20 could be made as follows:

```
/dev/fd0>//20/dev/fd0
```

In both cases above, the alias redirection could be bypassed and the remote resource could be directly named as:

```
//20/dev/spool OR //20/dev/fd0
```

Relative pathnames

Pathnames need not start with a single or a double slash. In such cases, the path is considered *relative* to the current working directory. QNX 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 single slash versus starting with a network root.

Current working directory

If the current working directory has a leading double slash (network root), it is said to be *specific* and locked to the pathname space of the specified node. If instead it has a single leading slash, the default network root is prepended.

For example, this command:

```
cd //18/
```

is an example of the first (specific) form, and would lock future relative pathname evaluation to be on node 18, no matter what your default network root happens to be. Subsequently entering `cd dev` would put you in `//18/dev`.

On the other hand, this command:

```
cd /
```

would be of the second form, where the default network root would affect the relative pathname resolution. For example, if your default network root were `//9`, then entering `cd dev` would put you in

//9/dev. Because the current working directory doesn't start with a node override, the default network root is prepended to create a fully specified network pathname.

This really isn't as complicated as it may seem. Typically, network roots (*//node/*) are *not* specified, and everything you do will simply work within your namespace (defined by your default network root). Most users will log in, accept the normal default network root (i.e. the namespace of their own node), and work within that environment.

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

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

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

For more information about symbolic links and `..` references, see Chapter 5, The Filesystem Manager.



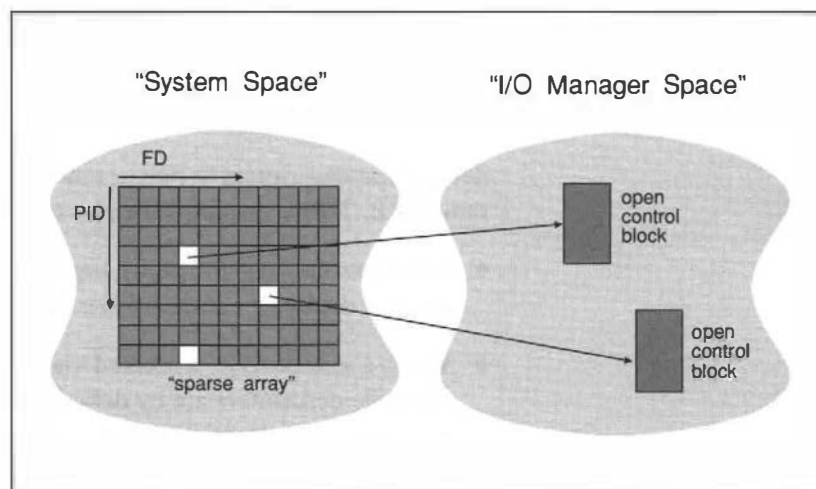
To display a fully resolved network pathname, you can use the `fullpath` utility.

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 manager. (Note that the *Sendfd()* kernel call is used within library routines to direct the request.)

The file descriptor namespace, unlike the pathname space, is completely local to each process. The manager uses the combination of a PID and FD 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 I/O manager.

The following diagram shows an I/O manager taking some PID, FD pairs and mapping them to OCBs.

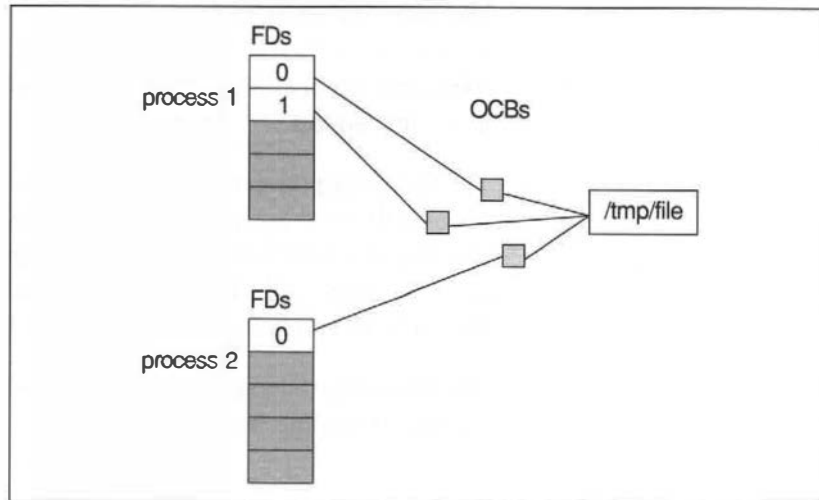


The PID and FD map to an OCB of an I/O Manager.

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.



Process 1 opens the file `/tmp/file` twice. Process 2 opens the same file once.

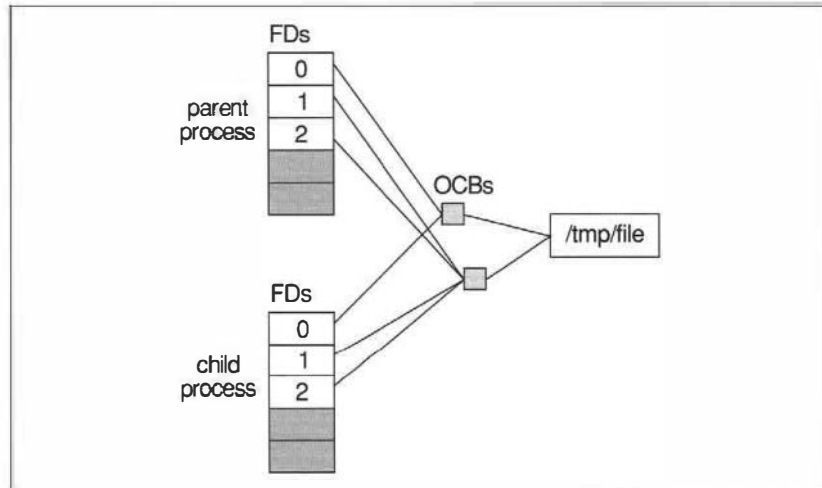
Several file descriptors in one or more processes can refer to the same OCB. This is accomplished by two means:

- A process may use the `dup()`, `dup2()`, or `fcntl()` C functions to create a duplicate file descriptor that refers to the same OCB.
- When a new process is created via `fork()`, `spawn()`, or `exec()`, 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. The process then creates a child that inherits all open files.



A process opens a file twice, then is duplicated via `dup()` to get a third FD. Its child process will inherit these three file descriptors.

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.

The Filesystem Manager

In this Chapter...

Introduction

What is a file?

Regular files and directories

Links and inodes

Symbolic links

Pipes and FIFOs

Filesystem Manager performance

Filesystem robustness

Raw volumes

Key components of a QNX partition

The DOS Filesystem Manager

Introduction

The Filesystem Manager (**Fsys**) provides a standardized means of storing and accessing data on disk subsystems. **Fsys** is responsible for handling all requests to open, close, read, and write files.

What is a file?

In QNX, a *file* is an object that can be written to, read from, or both. QNX implements six types of files; five of these are managed by **Fsys**:

- **Regular files**—consist of randomly accessible sequences of bytes and have no other predefined structure
- **Directories**—contain the information needed to locate regular files; they also contain status and attribute information for each regular file
- **Symbolic links**—contain a pathname to a file or directory that is to be accessed in place of the symbolic link file; these files are often used to provide multiple paths to a single file
- **Pipes and FIFOs**—serve as I/O channels between cooperating processes
- **Block special files**—refer to devices, such as disk drives, tapes, and disk drive partitions; these files are normally accessed in a manner that hides the hardware characteristics of the device from applications

All of these filetypes are described in detail in this chapter. The sixth filetype, the *character special file*, is managed by the Device Manager.

Date and time stamps

Fsys maintains four different times for each file. These are:

- date of last access (read)
- date of last write
- date of last modification
- date of creation (unique to QNX)

File access

Access to regular files and directories is controlled by mode bits stored in the file's inode (inodes are discussed in the section on "Links and inodes"). These bits permit read, write, and execute capability based on effective user and group IDs. There are three access qualifiers:

- user only
- group only
- others

A process can also run with the user ID or group ID of a file rather than those of its parent process. The mechanism that allows this is referred to as *setuid* (set user ID on execution) and *setgid* (set group ID on execution).

Regular files and directories

Regular files

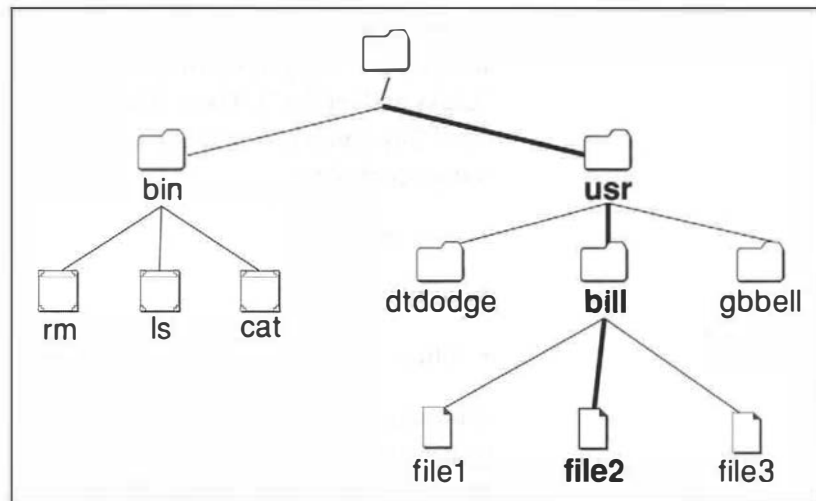
QNX views a *regular file* as a randomly accessible sequence of bytes that has no other predefined internal structure. Application programs are responsible for understanding the structure and content of any specific regular file.

Regular files constitute the majority of files found in *filesystems*. Filesystems are supported by the Filesystem Manager and are implemented on top of the block special files that define disk partitions (partitions are described in the section on "Raw volumes").

Directories

A *directory* is a file that contains *directory entries*. Each directory entry associates a *filename* with a file. A filename is the symbolic name that lets you identify and access a file. A file may be identified by more than one filename (see the sections on "Links and inodes" and "Symbolic links").

The following diagram shows how the directory structure is navigated to locate the file `/usr/bill/file2`.



The path through the QNX directory structure to the file `usr/bill/file2`.

Directory operations

Although a directory behaves much like a standard file, the Filesystem Manager imposes some restrictions on the operations you can perform on a directory. Specifically, you can't open a directory for writing, nor can you link to a directory with the `link()` C function.

Reading directory entries

To read directory entries, you use a set of POSIX-defined C functions that provide access to directory entries in an OS-independent fashion. These functions include:

```

opendir()
readdir()
rewinddir()
closedir()

```

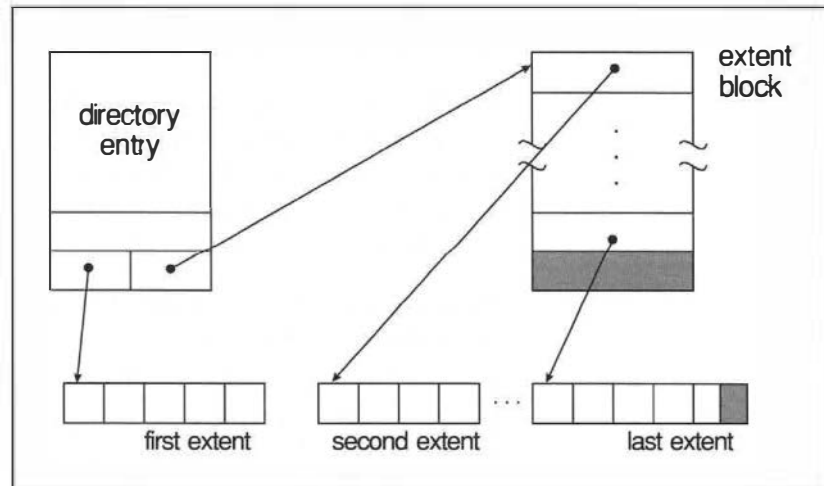
Since QNX directories are simply files that contain “known” information, you can also read directory entries directly using the `open()` and `read()` C functions. This technique isn't portable, however—the format of directory entries varies from operating system to operating system.

Extents

Where extents are stored

In QNX, regular files and directory files are stored as a sequence of *extents*. An extent is a contiguous set of blocks on disk.

Files that have only a single extent store the extent information in the directory entry. But if more than one extent is needed to hold the file, the extent location information is stored in one or more *linked extent* blocks. Each extent block can hold location information for up to 60 extents.



A file consisting of multiple consecutive regions on a disk—called extents in QNX.

Extending files

When the Filesystem Manager needs to extend a file whose last extent is full, it first tries to extend the last extent, even if only by one block. But if the last extent can't be extended, a new extent is allocated to extend the file.

To allocate new extents, the Filesystem Manager uses a “first fit” policy. A special table in the Filesystem Manager contains an entry for each block represented in the `/.bitmap` file (this file is described in the section on “Key components of a QNX partition”). Each of these entries defines the largest contiguous free extent in the area defined by its corresponding block. The Filesystem Manager chooses the first entry in this table large enough to satisfy the request for a new extent.

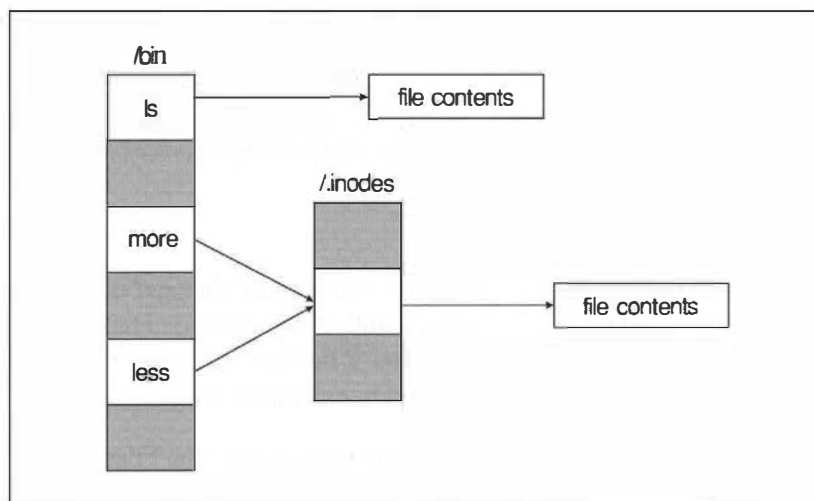
Links and inodes

In QNX, file data can be referenced by more than one name. Each filename is called a *link*. (There are actually two kinds of links: hard links, which we refer to simply as “links,” and symbolic links. Symbolic links are described in the next section.)

In order to support links for each file, the filename is separated from the other information that describes a file. The non-filename information is stored in a structure called an *inode*.

If a file has only one link (i.e. one filename), the inode information (i.e. the non-filename information) is stored in the directory entry for the file. If the file has more than one link, the inode is stored as a record in a special file named `/.inodes`, as are the file’s directory entry points to the inode record.

Note that you can create a link to a file only if the file and the link are in the same filesystem.



The same file is referenced by two links named “more” and “less.”

There are two other situations in which a file can have an entry in the `/ .inodes` file:

- If a file's filename is longer than 16 characters, the inode information is stored in the `/ .inodes` file, making room for a 48-character filename in the directory entry.
- If a file has had more than one link and all links but one have been removed, the file continues to have a separate `/ .inodes` file entry. This is done because the overhead of searching for the directory entry that points to the inode entry would be prohibitive (there are no back links from inode entries to directory entries). You can correct this condition with `chksys`.

If you want to:	use the:
create links from within the Shell	<code>ln</code> utility
create links from within programs	<code>link()</code> C function

Removing links

When a file is created, it is given a *link count* of one. As links to the file are added, this link count is incremented; as links to the file are removed, the link count is decremented. The filespace isn't removed from disk until its link count goes to zero *and* all programs using the file have closed it. This allows an open file to remain in use, even though it has been completely unlinked.

If you want to:	use the:
remove links from within the Shell	<code>rm</code> utility
remove links from within programs	<code>remove()</code> or <code>unlink()</code> C functions

Directory links

You cannot create hard links to directories. However, directories have two hard-coded links:

- `.` "dot"
- `..` "dot dot"

The filename “dot” refers to the directory specified by its predecessor in a pathname; “dot dot” refers to the parent directory of its predecessor.

`/usr/home/fred/. /test` → `/usr/home/fred/test`

└──
Predecessor

`/usr/home/fred/.. /eric` → `/usr/home/eric`

└──┬──
└── *Predecessor*
└── *Parent of predecessor*

If there’s no predecessor, “dot” refers to the current directory. Also, the “dot dot” entry of “/” is simply “/”—you can’t go further up the path.

Symbolic links

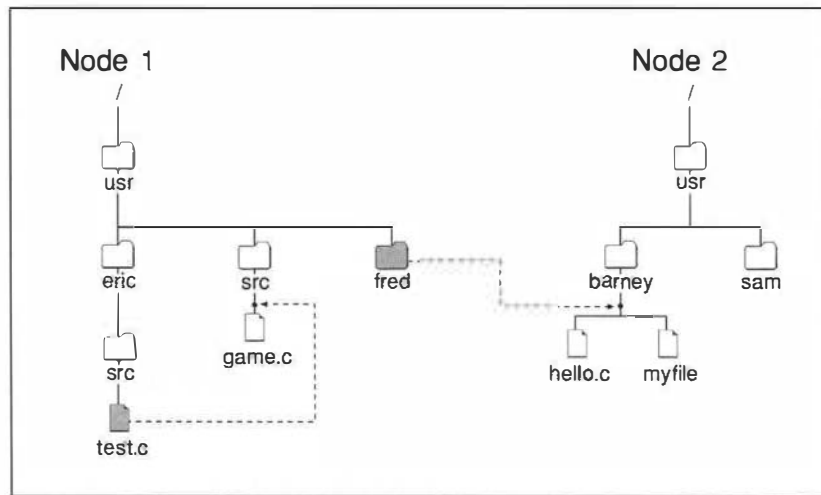
A *symbolic link* is a special file that has a pathname as its data. When the symbolic link is named in an I/O request—by *open()*, for example—the link portion of the pathname is replaced by the link’s “data” and the path is re-evaluated. Symbolic links are a flexible means of pathname indirection and are often used to provide multiple paths to a single file. Unlike hard links, symbolic links can cross filesystems and can also create links to directories.

In the following example, the directories `//1/usr/fred` and `//2/usr/barney` are linked even though they reside on different filesystems—they’re even on different nodes (see the following diagram). This couldn’t be done using hard links:

`//1/usr/fred` → `//2/usr/barney`

Note how the symbolic link and the target directory need not share the same name. In most cases, you use a symbolic link for linking one directory to another directory. However, you can also use symbolic links for files, as in this example:

`//1/usr/eric/src/test.c` → `//1/usr/src/game.c`



Two nodes using symbolic links.

If you want to:	use the:
create symbolic links	ln utility (with -s option)
remove symbolic links*	rm utility
query whether a file is a symbolic link	ls utility

* Remember that removing a symbolic link acts only on the link, and not on the target.

Several functions operate directly on the symbolic link. For these functions, the replacement of the symbolic element of the pathname with its target is not performed. These functions include *unlink()* (which removes the symbolic link), *lstat()*, and *readlink()*.

Since symbolic links can point to directories, incorrect configurations can result in problems such as circular directory links. To recover from circular references, the system imposes a limit on the number of hops; this limit is defined as `{SYMLOOP_MAX}` in the `<limits.h>` file.

Pipes and FIFOs

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 Filesystem 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 its ends have closed.

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	the pipe symbol (“ ”)
create pipes from within programs	<i>pipe()</i> or <i>popen()</i> C functions



On diskless workstations, you can run the Pipe Manager (**Pipe**) in place of the Filesystem Manager when only pipes are required. The Pipe Manager is optimized for pipe I/O and may achieve better pipe throughput than the Filesystem Manager.

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 <i>FIFOs</i> from within the Shell	<code>mkfifo</code> utility
create <i>FIFOs</i> from within programs	<code>mkfifo()</code> C function
remove <i>FIFOs</i> from within the Shell	<code>rm</code> utility
remove <i>FIFOs</i> from within programs	<code>remove()</code> or <code>unlink()</code> function

Filesystem Manager performance

The Filesystem Manager has several features that contribute to high-performance disk access:

- elevator seeking
- buffer cache
- multi-threading
- client-driven priority
- temporary files
- ramdisks

Elevator seeking *Elevator seeking* minimizes the overall seek time required to write data to disk. Outstanding write requests are ordered such that they can all be written with one sweep of the disk head assembly, from the lowest to the highest disk address.

Elevator seeking also has integrated enhancements to ensure that multi-sector writes are performed whenever possible.

Buffer cache The *buffer cache* is an intelligent buffer between the Filesystem Manager and the disk driver. The buffer cache attempts to store filesystem blocks in order to minimize the number of times the

Filesystem Manager has to access the disk. By default, the size of the cache is determined by total system memory, but you can specify a different size via an option to **Fsys**.

Read operations are synchronous. Write operations, on the other hand, are usually asynchronous. When the data enters the cache, the Filesystem Manager replies to the client process to indicate that the data is written. The data is then written to the disk as soon as possible, typically less than two seconds later.

The cache management software normally gives priority to read operations over write operations. This behavior is modified in two cases:

- The cache is nearly full—writes are given the same priority as reads until a reasonable number of blocks have been made available in the cache.
- Critical filesystem blocks such as bitmap blocks, directory blocks, extent blocks, and inode blocks are written immediately and synchronously to disk (these critical blocks bypass the normal write mechanism, including the elevator seeking).

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.

Multi-threading

The Filesystem Manager is a multi-threaded process. That is, it can manage several I/O requests simultaneously. This allows the Filesystem Manager to fully exploit potential parallelism since it can do both of the following:

- access several devices in parallel
- satisfy I/O requests from the buffer cache while other I/O requests that access physical disks are in progress

**Client-driven
priority**

The Filesystem Manager may have its priority driven by the priority of the processes that send it messages. When the Filesystem Manager receives a message, its priority is set to that of the process that sent the message. For more information, see “Process scheduling” in Chapter 2.

Temporary files

QNX has a performance option for opening temporary files that are written and then reread in a short period of time. For such files, the Filesystem Manager attempts to keep the data blocks in the cache and will write the blocks to disk only if absolutely necessary.

Ramdisks

The Filesystem Manager has an integrated ramdisk capability that allows up to 8M of memory to be used as a simulated disk. Since the Filesystem Manager uses highly efficient multi-part messaging, data moves from the ramdisk directly to the application buffers. The Filesystem Manager is able to bypass the buffer cache because the ramdisk is built in, not implemented as a driver. (For information on multi-part messaging, see Chapter 2, *The Microkernel*.) Because they eliminate the delays of physical hardware and don’t rely on the filesystem cache, ramdisks provide greater determinism in read/write operations than do hard disks.

Filesystem robustness

The QNX filesystem achieves high throughput without sacrificing reliability. This has been accomplished in several ways.

While most data is held in the buffer cache and written after only a short delay, critical filesystem data is written immediately. Updates to directories, inodes, extent blocks, and the bitmap are forced to disk to ensure that the filesystem structure on disk is never corrupt (i.e. the data on disk should never be internally inconsistent).

Sometimes all of the above structures must be updated. For example, if you move a file to a directory and the last extent of that directory is full, the directory must grow. In such cases, the order of operations has been carefully chosen such that if a catastrophic failure occurs with the operation only partially completed (e.g. a power failure), the

Filesystem recovery

filesystem, upon rebooting, would still be “sane.” At worst, some blocks may have been allocated, but not used. You can recover these for later use by running the **chkfsys** utility.

Even in the best systems, true catastrophes such as these may happen:

- Bad blocks may develop on a disk because of power surges or brownouts.
- A naive or malicious user with access to superuser privileges might reinitialize the filesystem (via the **dinit** utility).
- An errant program (especially one run in a non-QNX environment) may ignore the disk partitioning information and overwrite a portion of the QNX partition.

So that you can recover as many of your files as possible if such events ever occur, unique “signatures” have been written on the disk to aid in the automatic identification and recovery of the critical filesystem pieces. The inodes file (**/ . inodes**), as well as each directory and extent block, all contain unique patterns of data that the **chkfsys** utility can use to reassemble a truly damaged filesystem.

For details on filesystem recovery, see the documentation for the **chkfsys** utility.

Raw volumes

The Filesystem Manager manages *block special files*. These files define *disks* and *disk partitions*.

Disks and disk subsystems

QNX considers each physical disk on a computer to be a *block special file*. As a block special file, a disk is viewed by a QNX filesystem as a sequential set of blocks, each 512 bytes in size, regardless of the size and capacities of the disk. Blocks are numbered, beginning with the first block on the disk (block 1).

Because each disk is a block special file, it can be opened as an entity for raw-level access using standard POSIX C functions such as *open()*, *close()*, *read()* and *write()*. At the level of a block special file that defines an entire disk, QNX makes absolutely no assumptions about any data structure that may exist on the disk.

A computer running QNX may have one or more *disk subsystems*. Each disk subsystem consists of a controller and one or more disks. You start a device driver process for each disk subsystem that is to be managed by the Filesystem Manager.

OS partitions

QNX complies with a de facto industry standard that allows a number of operating systems to share the same physical disk. According to this standard, a partition table can define up to four primary partitions on the disk. The table is stored on the first disk block.

Each partition must be given a “type” recognized by the operating system prepared to handle that partition. The following list shows the partition types that are currently used:

Type:	Operating System:
1	DOS (12-bit FAT)
4	DOS (16-bit FAT)
5	DOS Extended Partition
6	DOS 4.0 large partitions (>32M)
7	QNX 1.x and 2.x (“ qnx ”)
7	OS/2 HPFS
8	QNX 1.x and 2.x (“ qny ”)
9	QNX 1.x and 2.x (“ qnz ”)
77	QNX 4.x
78	QNX 4.x
79	QNX 4.x
99	UNIX

If you want more than one QNX 4.x partition on a single physical disk, you would use type 77 for your first QNX partition, type 78 for your second QNX partition, and type 79 for your third QNX partition.

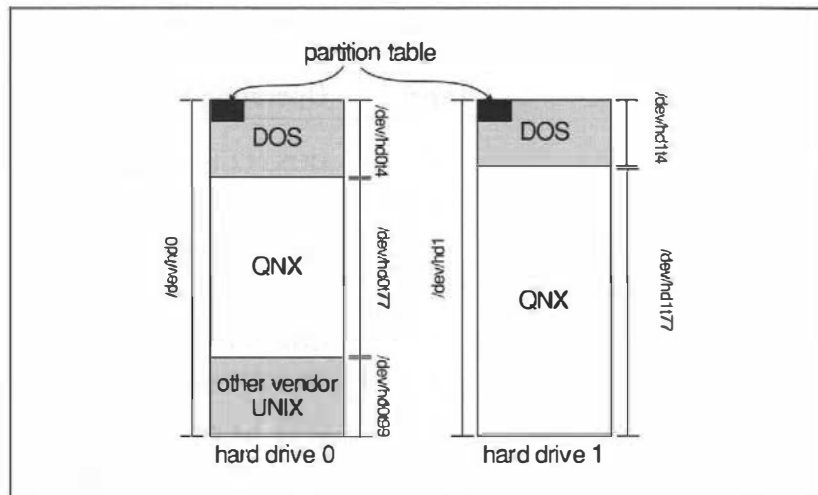
You can use other types for your second and third partitions, but 78 and 79 are preferred. To mark any of these partitions as bootable, you use the **fdisk** utility.

At boot time, the QNX boot loader (optionally installed by **fdisk**) lets you override the default boot partition selection in the partition table.

You can use the **fdisk** utility to create, modify, or delete partitions.

Because QNX treats each partition on a disk as a block special file, you can access either of the following:

- the complete disk—disregarding partitions—as a block special file
- a single partition as a block special file; this block special file will be a subset of the block special file that defines the entire disk



Two physical disk drives. The first drive contains DOS, QNX, and UNIX partitions. The second has DOS and QNX partitions.

Defining block special files

The names of all block special files are placed in the prefix tree for the computer where the block special files reside (the prefix tree is described in Chapter 3, I/O Namespace). When a device driver for a disk subsystem is started, the Filesystem Manager automatically registers prefixes that define a block special file for each physical drive attached to the disk subsystem. The `mount` utility is then used to register a prefix for a block special file for each partition on the disk subsystem.

For example, let's say you have a standard Western Digital disk controller with two attached drives. On one drive, you want to mount a DOS partition, a QNX partition, and a UNIX partition. On the other drive, you want to mount a DOS partition and a QNX partition.

The Filesystem Manager would define the block special files `/dev/hd0` and `/dev/hd1` for the two drives on the controller where the driver was started.

You would then use the `mount` utility to define a block special file for each partition. For example:

```
mount -p /dev/hd0 -p /dev/hd1
```

would yield the following block special files:

OS partition:	Block special file:
DOS partition on drive <code>hd0</code>	<code>/dev/hd0t4</code>
QNX partition on drive <code>hd0</code>	<code>/dev/hd0t77</code>
UNIX partition on drive <code>hd0</code>	<code>/dev/hd0t99</code>
DOS partition on drive <code>hd1</code>	<code>/dev/hd1t4</code>
QNX partition on drive <code>hd1</code>	<code>/dev/hd1t77</code>

Note that the `tn` convention is used to refer to disk partitions used by certain operating systems. For example a DOS partition is `t4`, a UNIX partition is `t99`, etc.

Mounting a filesystem

You typically mount a QNX filesystem on a block special file. To mount a filesystem, you again use the **mount** utility—it lets you specify the prefix that identifies the filesystem. For example:

```
mount /dev/hd0t77 /
```

mounts a filesystem with a prefix of **/** on the partition defined by the block special file named **hd0t77**.



If a disk has been partitioned, you must mount a partition block special file (e.g. **/dev/hd0t77**) that defines a QNX 4.x partition, not the base block special file that defines the entire raw disk (e.g. **/dev/hd0**). If you attempt to mount the base block special file for the entire disk, you'll get a "corrupt filesystem" message when you try to access the filesystem.

Unmounting a filesystem

To unmount a filesystem you use the **umount** utility. For example, the following command will unmount the filesystem on your primary QNX partition:

```
umount /dev/hd0t77
```

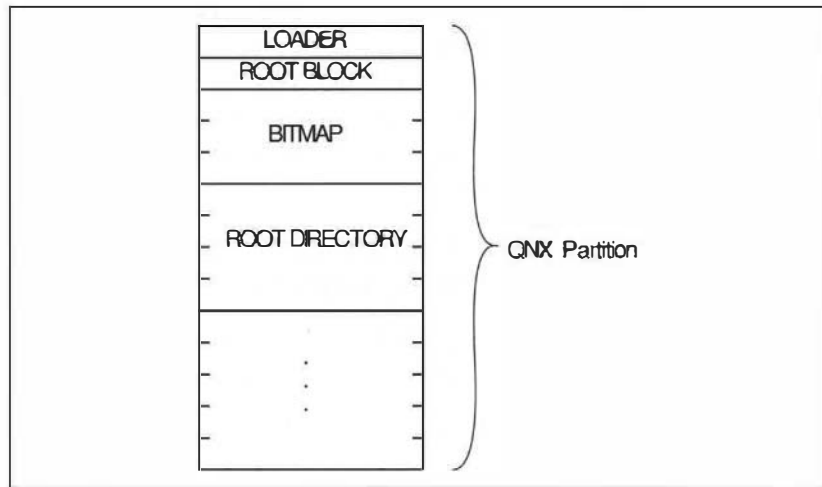
Once a filesystem is unmounted, files on that partition are no longer accessible.

Key components of a QNX partition

Several key components found at the beginning of every QNX partition tie the filesystem together:

- loader block
- root block
- bitmap
- root directory

These structures are created when you initialize the filesystem with the `dinit` utility.



Structure of a QNX filesystem within a disk partition.

Loader block

This is the first physical block of a disk partition. This block contains the code that is loaded and then executed by the BIOS of the computer to load an operating system from the partition. If a disk hasn't been partitioned (e.g. as in a floppy diskette), this block is the first physical block on the disk.

Root block

The root block is structured as a standard directory. It contains inode information for these four special files:

- the *root directory* of the filesystem (*/*)
- */.inodes*
- */.boot*
- */.altboot*

The files */.boot* and */.altboot* contain images of the operating system that can be loaded by the QNX bootstrap loader.

Normally, the QNX loader loads the OS image stored in the */.boot* file. But if the */.altboot* file isn't empty, you'll be given the option to load the image stored in the */.altboot* file.

Bitmap

To allocate space on a disk, QNX uses a *bitmap* stored in the */.bitmap* file. This file contains a map of all the blocks on the disk, indicating which blocks are used. Each block is represented by a bit. If the value of a bit is 1, its corresponding block on the disk is being used.

Root directory

The root directory of a partition behaves as a normal directory file with two exceptions:

- Both “dot” and “dot dot” are links to the same inode information, namely the root directory inode in the root block.
- The root directory always has entries for the */.bitmap*, */.inodes*, */.boot*, and */.altboot* files. These entries are provided so programs that report information on filesystem usage will see the entries as normal files.

The DOS Filesystem Manager

In QNX, the I/O namespace is managed through prefixes that direct file requests to the appropriate manager process. A process that takes advantage of this is the DOS Filesystem Manager (**Dosfsys**). **Dosfsys** administers the **/dos** namespace prefix and presents DOS drives within the QNX namespace as “guest” filesystems.

Dosfsys provides transparent access to DOS disks, so you can treat DOS filesystems as though they were QNX filesystems. This transparency allows processes to operate on DOS files without any special knowledge or work on their part. Standard I/O library functions, such as *open()*, *close()*, *read()*, and *write()*, operate identically for a file on a DOS partition as for a file on a QNX partition. For example, to copy a file from your QNX partition to your DOS partition, you would simply enter:

```
cp /usr/luc/file.dat /dos/c/file.dat
```

The pathname of your DOS drive **c**: under QNX.

The **cp** command contains no special code to detect whether the file it is copying is located on a DOS disk. Other commands also work with equal transparency (e.g. **cd**, **ls**, **mkdir**).

If there's no DOS equivalent to a QNX feature, such as *mkfifo()* or *link()*, an appropriate error code (i.e. *errno*) is returned by **Dosfsys**.

Dosfsys works both with floppies and with hard disk partitions. All of the low-level disk access that **Dosfsys** requires is done using standard functions provided by the Filesystem Manager. Thus, with no low-level code, **Dosfsys** is able to integrate a seamless interface between QNX applications and a DOS filesystem.



If you wish to run DOS executables under QNX, an optional package, **Rundos**, is available.

Chapter 6

The Device Manager

In this Chapter...

Introduction

Device services

Edited input mode

Raw input mode

Device drivers

The QNX console

Serial devices

Parallel devices

Device subsystem performance

Introduction

The QNX Device Manager (**Dev**) is the interface between processes and *terminal devices*. These terminal devices are located in the I/O namespace with names starting with **/dev**. For example, a console device on QNX would have a name such as:

/dev/con1

Device services

QNX programs access terminal devices using the standard *read()*, *write()*, *open()*, and *close()* functions. A terminal device is presented to a QNX process as a bidirectional stream of bytes that can be read or written by the process.

The Device Manager regulates the flow of data between an application and the device. Some processing of this data is performed by **Dev** according to parameters in a *terminal control structure* (called *termios*), which exists for each device. Users can query and/or change these parameters by using the **stty** utility; programs can use the *tcgetattr()* and *tcsetattr()* functions.

The *termios* parameters control low-level functionality such as:

- line-control discipline (including baud rate, parity, stop bits, and data bits)
- echoing of characters
- input line editing
- recognizing, and acting on, breaks and hangups
- software and hardware flow control
- translation of output characters

The Device Manager also provides a set of auxiliary services available to processes for managing terminal devices. The following table summarizes some of these services.

A process can:	via this C function:
perform timed read operations	<i>dev_read()</i> or <i>read()</i> + <i>tcsetattr()</i>
asynchronously notify a process of data available on one or more input devices	<i>dev_arm()</i>
wait for output to be completely transmitted	<i>tcdrain()</i>
send breaks across a communications channel	<i>tcsendbreak()</i>
disconnect a communications channel	<i>tcdropline()</i>
insert input data	<i>dev_insert_chars()</i>
perform non-blocking reads and writes	<i>open()</i> and <i>fcntl()</i> (O_NONBLOCK mode)

Edited input mode

The most significant mode of device processing is controlled by the ICANON bit in the *termios* control structure. If this control bit is set, the Device Manager performs line-editing functions on received characters. Thus, only when a line is “entered”—typically when a carriage return (CR) is received—will the processed data be made available to application processes. This mode of operation is referred to as *edited*, *canonical*, or sometimes *cooked* mode.

Most non full-screen applications run in *edited* mode. The Shell is a typical example.

The following table shows several special control characters that may be set in the *termios* control structure to define how **Dev** performs this editing.

Dev will:	when it receives:
move the cursor one character to the left	LEFT
move the cursor one character to the right	RIGHT
move the cursor to the beginning of the line	HOME
move the cursor to the end of the line	END
rub out the character to the left of the cursor	ERASE
delete the character at the current cursor position	DEL
rub out the entire input line	KILL
erase the current line and recall a previous line	UP
erase the current line and recall the next line	DOWN
toggle between insert mode and typeover mode (every new line starts in typeover mode)	INS

Line-editing characters vary from terminal to terminal. The QNX console always starts out with a full set of editing keys defined.

If a terminal is connected to QNX via a serial channel, it's necessary 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 a VT100 terminal connected to a serial port (called **/dev/ser1**), you would use the following statement to extract the appropriate editing keys from the *terminfo* database and apply them to **/dev/ser1**:

```
stty term=vt100 </dev/ser1
```

If, instead, you had a modem connected to that serial port, which in turn was connected to another QNX system running the **qtalk** utility, you'd want to set the line-editing keys as follows:

```
stty term=qnx </dev/ser1
```

Raw input mode

When ICANON isn't set, the device is said to be in *raw* mode. In this mode, no input editing is performed, and any received data is made immediately available to QNX processes.

Full-screen programs and serial communications programs are examples of QNX applications that put a device in *raw* mode.

When reading from a *raw* device, the application is capable of specifying under what conditions an input request is to be satisfied. The criteria used for accepting raw input data are based on two members of the *termios* control structure: MIN and TIME. The application can specify a further qualifier of accepting input data when it issues its read request via *dev_read()*. This qualifier, TIMEOUT, is useful when writing protocols or realtime applications. Note that TIMEOUT is always $\emptyset$ for *read()*.

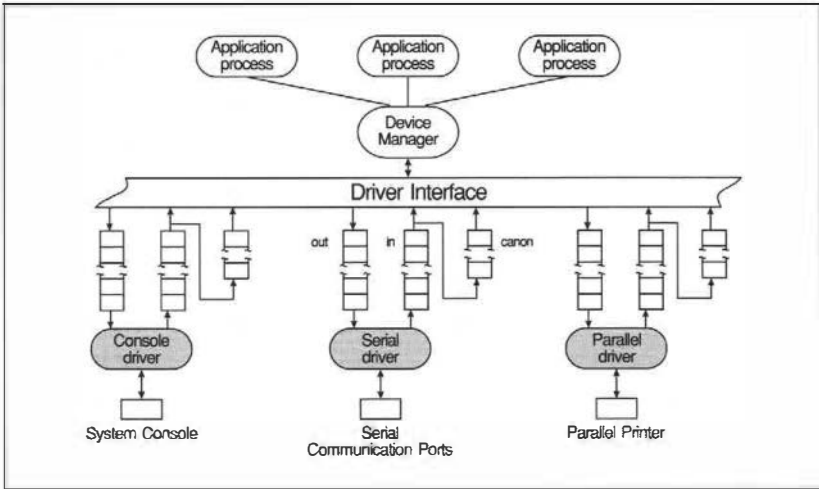
When a QNX process issues a read request for *n* bytes of data, these three parameters define when that read request is to be satisfied:

MIN	TIME	TIMEOUT	Description
$\emptyset$	$\emptyset$	$\emptyset$	Return immediately with as many bytes as are currently available (up to <i>n</i> bytes).
M	$\emptyset$	$\emptyset$	Return with up to <i>n</i> bytes only, when at least M bytes are available.
$\emptyset$	T	$\emptyset$	Return with up to <i>n</i> bytes when at least one byte is available, or T $\times$.1 second has expired.
M	T	$\emptyset$	Return with up to <i>n</i> bytes when either M bytes are available or at least one byte has been received and the interbyte time between any subsequently received characters exceeds T $\times$.1 second.

MIN	TIME	TIMEOUT	Description
Ø	Ø	t	RESERVED.
M	Ø	t	Return with up to n bytes when t × .1 second has expired, or M bytes are available.
Ø	T	t	RESERVED.
M	T	t	Return with up to n bytes when M bytes are available, or t × .1 second has expired and no characters are received, or at least one byte has been received and the interbyte time between any subsequently received characters exceeds T × .1 second.

Device drivers

The following illustration shows a typical QNX device subsystem.



The Device Manager process (**Dev**) manages the flow of data to and from the QNX application processes. The hardware interface is managed by individual *driver* processes. Data flows between **Dev** and its *drivers* through a set of shared memory queues* for each *terminal device*.

Three queues are used for each device. Each queue is implemented using a first-in, first-out mechanism. A control structure is also associated with each queue.

Received data is placed into the raw input queue by the driver and is consumed by **Dev** only when application processes request data. Interrupt handlers within drivers typically call a trusted library routine within **Dev** to add data to this queue—this ensures a consistent input discipline and greatly minimizes the responsibility of the driver.

Dev places output data into the output queue; the data is consumed by the driver as characters are physically transmitted to the device. **Dev** calls a trusted routine within the driver process 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, **Dev** implements full *write-behind* for all *terminal devices*. Only when the output buffers are full will **Dev** cause a process to block while writing.

The canonical queue is managed entirely by **Dev** 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 all these queues are configurable by the system administrator; the only restriction is that the sum total of all three queues can't exceed 64K. Default values are usually more than adequate to handle most hardware configurations, but you can “tune” these either to reduce overall system memory requirements or to accommodate unusual hardware situations.

* Since shared memory queues are used, it's necessary that **Dev** and all of its drivers reside on the same physical CPU. The advantage, of course, is increased performance.

Device control

Device drivers simply add received data to the raw input queue or consume and transmit data from the output queue. **Dev** decides when (and if) output transmission is to be suspended, how (and if) received data is echoed, etc.

To ensure good interactive response to input events, **Dev** must run at a reasonably high priority. **Dev** typically has very little actual work to do when it does run, so it seldom impedes overall system performance.

The drivers themselves are just like any other QNX process—they can run at different priorities according to the nature of the hardware they’re serving.

Low-level device control is implemented with a far call into an *ioctl* entry point within each driver. A common set of *ioctl* commands are supported by most drivers used directly by **Dev**. Device-specific *ioctl* commands can also be sent through **Dev** to the drivers by QNX processes (via the *qnx_ioctl()* C function).

The QNX console

System consoles are managed by the **Dev.con** driver process. The display adapter and the screen, plus the system keyboard, are collectively referred to as the *console*.

QNX permits multiple sessions to be run concurrently on consoles by means of *virtual consoles*. The **Dev.con** console driver process typically manages more than one set of I/O queues to **Dev**, which are made available to user processes as a set of *terminal 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* 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.

Console-specific functions

In addition to implementing the standard QNX Terminal (as defined in the *User's Guide*), the console driver also provides a set of console-specific functions that let application processes communicate via messages directly to a console driver process. Communication is established with the `console_open()` C function. Once communication is established, a QNX process has access to the following capabilities:

A process can:	via this C function:
read directly from the console screen	<code>console_read()</code>
write directly to the console screen	<code>console_write()</code>
be asynchronously notified of significant events*	<code>console_arm()</code>
control the console size	<code>console_size()</code>
switch the visible console	<code>console_active()</code>
* e.g. display data has changed, cursor has moved, display size has changed, visible console has changed, etc.	

The QNX console driver runs as a normal QNX process. Input characters from the keyboard are mapped by the keyboard interrupt handler and placed directly into the input queue. Output data is consumed and displayed by **Dev.con** while it is executing as a process.

Serial devices

Serial communication channels are managed by the **Dev.ser** driver process. This driver can manage more than one physical channel; it provides terminal devices with names such as **/dev/ser1**, **/dev/ser2**, etc.

When you start **Dev.ser**, you can specify command-line arguments that determine which—and how many—serial ports are installed. To see what serial ports may be available on a QNX system, use the **ls** utility:

```
ls /dev/ser*
```

Dev.ser is an example of a purely interrupt-driven I/O server. After initializing the hardware, the process itself goes to sleep. Received interrupts place input data directly into the input queue. The first output character on an idle channel is transmitted to the hardware when **Dev** issues the first kick call into the driver. Subsequent characters are transmitted by the appropriate interrupt being received.

Parallel devices

Parallel printer ports are managed by the **Dev.par** driver process. When you start **Dev.par**, you specify a command-line argument that determines which parallel port is installed. To see if a parallel port is available on a QNX system, use the **ls** utility:

```
ls /dev/par*
```

Dev.par is an output-only driver, so it has no input or canonical input queues. You can configure the size of the output buffer with a command-line argument when you start **Dev.par**. The output buffer can be made very large if you wish to provide a form of software print buffer.

Dev.par is an example of a completely non-interrupt I/O server. The parallel printer process normally remains RECEIVE-blocked, waiting for data to appear in its output queue and a kick from **Dev**. When data is available to print, **Dev.par** runs in a busy-wait loop (at relatively low adaptive priority), while waiting for the printer hardware to accept characters. This low priority busy-wait loop ensures that overall system performance isn't affected, yet on the average produces the maximum possible throughput to the parallel device.

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 now be satisfied, will the interrupt handler schedule **Dev** to run. In all other cases, the interrupt simply returns. Moreover, if **Dev** is already running, no scheduling takes place.
- When a read operation is satisfied, **Dev** 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 QNX—result in a very lean input model.

The Network Manager

In this Chapter...

Introduction

Network Manager responsibilities

Microkernel/Network Manager interface

Network drivers

Node and network identifiers

Choosing a network

Introduction

The Network Manager (**Net**) gives QNX users a seamless extension of the operating system's powerful messaging capabilities. Communicating directly with the Kernel, the Network Manager enhances QNX's message-passing IPC by efficiently propagating messages to remote machines. In addition, the Network Manager offers two advanced features:

- increased throughput via load balancing
- fault tolerance via redundant connectivity

Network Manager responsibilities

The Network Manager is responsible for propagating the QNX messaging primitives across a local area network. The standard messaging primitives used in local messaging are used *unmodified* in remote messaging. In other words, there's no special "network" *Send()*, *Receive()*, or *Reply()*.

An independent module

The Network Manager does *not* have to be built into the operating system image. It may be started and stopped at any time to provide or remove network messaging capabilities.

When the Network Manager starts, it registers with the Process Manager and Kernel. This activates existing code within the two that interfaces to the Network Manager. This means that network messaging and remote process creation are not just a layer added onto the operating system. Network messaging is integrated into the very heart of the messaging and process management primitives themselves. This deep integration at the lowest level gives QNX its network transparency and qualifies it as a fully distributed operating system. Since applications access all services via messaging, and since the Network Manager allows messages to flow transparently on the network, QNX nodes perform together as a single, logical computer.

Since the Network Manager and Kernel are distinct, the Kernel can achieve independence from the network hardware, and non-networked machines can benefit from reduced code requirements.

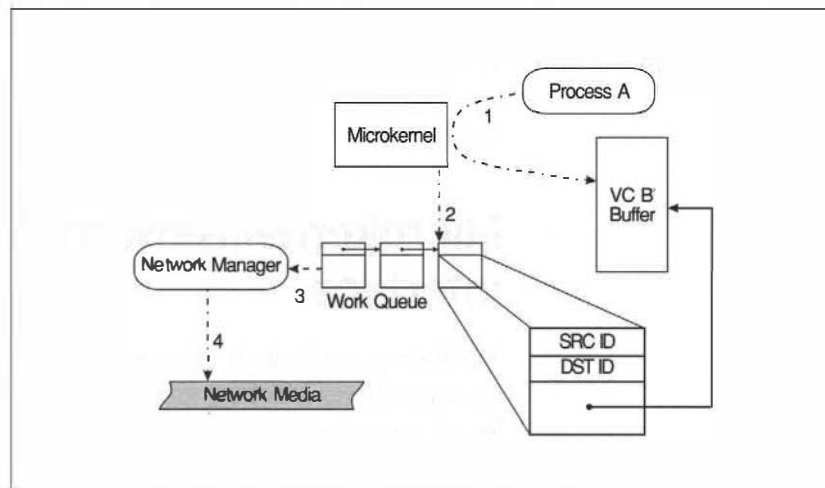
Microkernel/Network Manager interface

The Kernel and Process Manager interface to the Network Manager via a special nonblocking memory queue. This queue is essentially a list of transmissions to be performed by the Network Manager. The entries contain all the information for a particular operation (e.g. *Send()*, *Reply()*, VC creation, remote signal propagation, etc.).

Another resource the operating system uses to provide transparent messaging is the *virtual circuit buffer*. Allocated when a process creates a VC, virtual circuit buffers hold the data during the completion of a messaging transaction to another node.

The following diagrams outline the flow of data and control for sending and receiving remote messages.

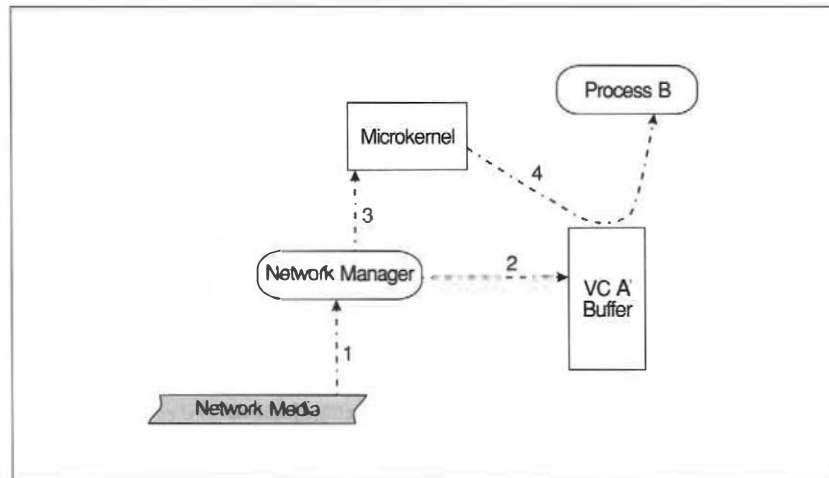
In the case of a *Send()* or *Reply()* to a remote node, the following events occur:



*A process issues a **Send()** or **Reply()** to a remote node.*

- 1) The process calls *Send()* or *Reply()* and the Kernel copies the data from the process's data space into the virtual circuit buffer associated with the VC specified by the *Send()* or *Reply()*.
- 2) The Kernel enqueues to the Network Manager a time-ordered queue entry identifying the sender, the remote receiver, and pointers to the data in the virtual circuit buffer. If the queue was previously empty, the Network Manager's proxy is triggered to let it know new work has arrived.
- 3) The Network Manager dequeues the queue entry.
- 4) The Network Manager begins transmission on the network media. The manager is responsible for delivery.

In the case of a signal propagation or VC creation, the Process Manager rather than the Kernel would enqueue a control packet. Nevertheless, the Network Manager would transmit the packet to its destination.



A process receives a remote `Send()` or `Reply()`.

Assuming the remote node will have sent a message as previously described, the following events occur on the node receiving the message:

- 1) Network data arrives on the media.
- 2) The Network Manager copies the message data into the appropriate virtual circuit buffer.
- 3) The Network Manager uses a private kernel call to inform the Kernel that the reception is complete.
- 4) The Kernel copies the data from the virtual circuit buffer into the process's buffer (assuming it was `RECEIVE-` or `REPLY-`blocked on this virtual circuit).

Any control packets the Network Manager receives are forwarded immediately to the Process Manager via the standard `Send()` primitive. These control packets are used for signal propagation and VC creation.

Network drivers

Like the Filesystem Manager and the Device Manager, the Network Manager contains no hardware-specific code. This functionality is provided by network card drivers. The Network Manager can support multiple network drivers at one time. Each driver typically supports a single network card. You may have drivers/cards of the same type or different types—for example, two Ethernet drivers/cards or perhaps an Ethernet driver/card and an Arcnet driver/card.

Shared memory queues provide the interface between the Network Manager and drivers. This interface has been designed to obtain the maximum performance possible. The driver determines the appropriate protocol for the network media.

The driver is responsible for packetization, sequencing, and retransmission if reliable guaranteed data transmission to a remote physical node is requested. This is the default for all QNX messaging primitives. This design allows QNX to easily support new network hardware and protocols by writing or modifying only a network driver.

Node and network identifiers

Each node in a local area network is identified by two numbers—its physical node ID and its logical node ID.

Physical node ID

The physical node ID is determined by the hardware. Network cards communicate with each other by specifying the physical node ID of the remote node they wish to talk to. In the case of Ethernet and Token Ring, this represents a large number that is difficult for people—and utilities—to deal with. For example, each Ethernet and Token Ring card is shipped with a unique 48-bit physical node ID, conforming to the IEEE 802 standard. Arcnet cards, on the other hand, have only an 8-bit ID.

Physical node IDs entail a significant drawback: when interconnecting some networks, addresses may conflict (especially in the case of Arcnet) or be of a radically different format.

Logical node ID

To overcome the above problems with physical node IDs, each QNX node is given a logical node ID. All QNX processes deal with logical node IDs. The physical node IDs are hidden from processes running on QNX.

Logical node IDs simplify network and application licensing. They also make it easy for some utilities that may wish to poll the network using a simple loop, where the logical node ID goes from 1 to the number of nodes.

The mapping between logical and physical node IDs is done by the Network Manager. The driver is given a physical ID by the Network Manager when asked to transmit data to another node.

The logical node ID is typically assigned sequential numbers starting at 1. For example, a node with an Ethernet card may be given a logical node ID of 2, which is mapped to the physical node ID of `00:00:c0:46:93:30`.

The logical node ID must be unique for all nodes across all interconnected QNX networks.

Logical network ID

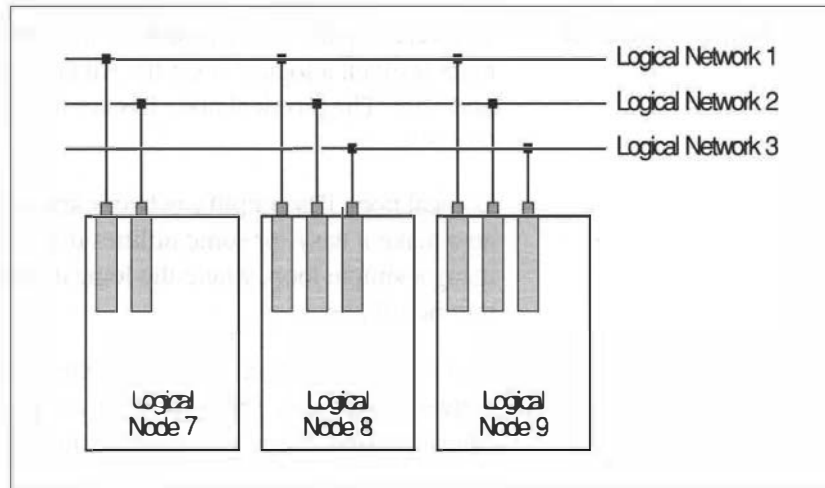
The network ID identifies a particular *logical network*. A logical network is any hardware that allows a network driver to directly communicate with a network driver on another node. This may be as simple as a serial port or as complex as an Ethernet network with bridges.

In the following diagram, node 7 has two network cards that allow the node to access nodes on logical networks 1 and 2. Nodes 8 and 9 both have three cards, connecting them to networks 1, 2, and 3.

Note that every logical node ID is unique across all three logical networks.



Logical network and logical node IDs are assigned by the system administrator. For more information, see “Network Installation” in the *User's Guide*.



Multiple physical networks happily coexist via logical networks.

Choosing a network

In the case where nodes are connected by more than one logical network, the Network Manager has a choice of which network to use when transmitting to a remote node. For example, in the above diagram, node 7 could transmit to node 8 using the driver attached to network 1 or network 2.

Load balancing

Network throughput is determined by a combination of the speed of the computer and the speed of the network. If your computer can provide data faster than the network can accept it, then the network will limit your throughput. For example, two 486 computers connected by an Arcnet network will be limited to a maximum of 2.5Mb/sec., which is the data rate provided by the hardware. If, however, you were to place two Arcnet cards in each computer and connect them with separate cables, the Network Manager could transmit data over both networks simultaneously. Under a heavy load, this would provide up to twice as much throughput as a single network.

The Network Manager will attempt to load balance by choosing an appropriate network driver. In our example above, if a transmission is in progress from node 7 to node 8 on network 1, and another transmission to node 8 is initiated on node 7, then network 2 will *automatically* be chosen to transmit the data.

Fault tolerance

When nodes are connected by two or more networks, there's more than a single path to use for communication. If a card in one network fails in a way that prevents any communication on that network, the Network Manager will automatically re-route all data through another network. This happens *on the fly* without any involvement on the part of the application software, and results in transparent network fault tolerance. If cables for the different networks are run using separate routes, you'll also be protected against an accidental cable cut.

You can also construct tandem systems in which two machines are connected by a fast network for normal operation and by a cheaper, slower network (e.g. a serial link) that remains on standby. If the first network fails, communication will continue, although throughput would naturally be reduced.

A

- access date (files) 66
- access to files 67
- adaptive scheduling 35
- alias prefixes 58
 - see also* I/O namespace
- .altboot file 84
- Arcnet drivers/cards 104
 - see also* network
- arithmetic operations, signal generated
 - by erroneous operation 24
- arming timers 50

B

- bad blocks on disk 78
- bitmap 84
- block special files 66, 78 – 82
 - defining 81
 - defining entire disk as 79
 - prefix tree where names are placed 81
 - registering prefixes 81
- blocked states
 - see* process states
- blocking signals 27

blocks

- bad blocks on disk 78
- loader block 83
- numbering 78
- root block 84
- used blocks on disk 84

booting

- .altboot file 84
- .boot file 84

breaks

- recognizing and acting on 88 – 89

- buffer cache (Filesystem Manager) 75

C

C functions

- _exit() 27
- abort() 24
- abstimer() 51
- access() 27
- alarm() 24, 27
- cfgetispeed() 27
- cfgetospeed() 27
- cfsetispeed() 27
- cfsetospeed() 27
- chdir() 27
- chmod() 27

chown() 27
close() 27, 79, 88
closedir() 68
console_active() 95
console_arm() 95
console_open() 95
console_read() 95
console_size() 95
console_write() 95
creat() 27
Creceive() 18
Creceivemx() 19
delay() 49
dev_arm() 89
dev_insert_chars() 89
dev_read() 89
dup() 27, 62
dup2() 27, 62
exec() 42 – 43
execle() 27
execve() 27
exit() 44
fcntl() 27, 62, 89
fcntrl() 63
fork() 27, 42 – 43
fstat() 27
getegid() 27
geteuid() 27
getgid() 27
getgroups() 27
getpgrp() 27
getpid() 27
getppid() 27
getprio() 33
getscheduler() 34
gettimer() 51
getuid() 27
kill() 22, 27
link() 27, 68, 71
longjmp() 26
lseek() 27, 61 – 62
mkdir() 27
mkfifo() 27, 75
mktimer() 50
open() 27, 31, 54, 58, 61, 68, 79, 88 – 89
opendir() 68
pathconf() 27
pause() 27
pipe() 27, 74
popen() 74
qnx_hint_attach() 52
qnx_hint_detach() 52
qnx_ioctl() 94
qnx_name_attach() 49
qnx_name_locate() 31, 49
qnx_osinfo() 48
qnx_pflags() 25, 37
qnx_prefix_getroot() 57
qnx_prefix_query() 55
qnx_prefix_setroot() 57
qnx_proxy_attach() 21
qnx_psinfo() 48
qnx_spawn() 45
qnx_spawn_options() 45
qnx_timerperiod() 51
qnx_vc_attach() 29
raise() 22
read() 27, 68, 79, 88 – 89
readdir() 68
Readmsg() 19
Readmsgmx() 19
Receive() 11, 14, 100
Receivemx() 19
reltimer() 51
remove() 71, 75
rename() 27
Reply() 11, 14, 16, 100 – 101
Replymx() 19
rewinddir() 68

- rmdir()* 27
- rmtimer()* 51
- Send()* 11, 14, 100 – 101
- Sendfd()* 61
- Sendmx()* 19
- setgid()* 27
- setpgid()* 27
- setprio()* 33
- setscheduler()* 34
- setsid()* 27
- setuid()* 27
- sigaction()* 26 – 27
- sigaddset()* 27
- sigdelset()* 27
- sigemptyset()* 27
- sigfillset()* 27
- sigismember()* 27
- siglongjmp()* 26
- signal()* 26 – 27
- sigpending()* 27
- sigprocmask()* 27
- sigsuspend()* 27
- sleep()* 27, 49
- spawn()* 42 – 43
- stat()* 27
- sysconf()* 27
- tcdrain()* 27, 89
- tcdropline()* 89
- tcflow()* 27
- tcflush()* 27
- tcgetattr()* 27, 88
- tcgetpgrp()* 27
- tcsendbreak()* 89
- tcsetattr()* 27, 88 – 89
- tcsetgrp()* 27
- time()* 27
- times()* 27
- Trigger()* 21
- tscendbreak()* 27
- umask()* 27
- uname()* 27
- unlink()* 27, 71, 75
- ustat()* 27
- utime()* 27
- wait()* 27, 45 – 46
- waitpid()* 27, 45 – 46
- write()* 27, 79, 88
- Writemsg()* 19
- Writemsgmx()* 19
- cache (Filesystem Manager) 75
- catching signals 26
- cd** command 59 – 60
 - see also* directories
- changing directories 59 – 60
 - see also* directories
- character special files 66
- characters
 - deleting 90
 - echoing 88
 - inserting 90
 - overwriting 90
 - translation of output characters 88
- child processes
 - creating 42
 - environment inherited from parent 43
 - running concurrently with their parents 44
 - zombies 45 – 46
 - see also* processes
- client-driven scheduling 36
- client/server model 16, 28, 36
- command interpreter (Shell)
 - see User's Guide and Utilities Reference*
- communications channel
 - disconnecting 89
 - sending breaks across 89
 - serial channel management 95
- conditional message reception 18
 - see also* message passing

- consoles
 - console-specific C functions 95
 - defined 94
 - multiple concurrent sessions 94
 - notification of significant events 95
 - reading screen 95
 - size, controlling 95
 - visible console, controlling 95
 - writing directly to screen 95
- controlling terminal
 - signal generated upon hangup 24
- cooperating processes
 - benefits of 48
 - communicating directly 11
 - FIFOs 75
 - messages, using 11
 - pipes 74
 - residing on different nodes 29
 - special considerations 48
 - synchronizing via message passing 13
 - see also* processes
- creating processes 42, 44
- creation date (files) 66
- Creceive()* (message-passing primitive) 19
- current working directory 59, 71
 - see also* directories
- cursor, moving 90

D

- data flow
 - between applications and devices 88, 93
 - between **Dev** and its drivers 93
- date and time stamps 66
- DEAD (process state) 46
- default network root 56, 59
 - changing 57
 - getting 57
 - setting 57

- deleting characters 90
- Dev.con** (driver process) 94
- Dev.par** (driver process) 96
- Dev.ser** (driver process) 95
- device control 94
- device drivers
 - see* drivers
 - see also* data flow
- Device Manager
 - auxiliary services for terminal devices 88
 - general responsibilities 88
- device names, creating 58
- devices
 - accessing across the network 55
 - device subsystem performance 97
 - parallel devices 96
 - serial devices 95
 - see also* terminal devices
- directories 66 – 67
 - changing 59 – 60
 - current working directory 59, 71
 - directory entries 67
 - directory entries, extent information 69
 - directory entries, reading 68
 - dot 71
 - dot dot 71
 - extents 69
 - filenames 67, 70 – 71
 - inodes 70 – 71
 - linking 68, 72 – 73
 - links 70 – 71
 - navigating directory structure 67
 - parent directory 71
 - pseudonyms 72 – 73
 - reading directory entries 67
 - root directory, special characteristics 84
 - writing 68
 - see also* files
- disconnecting communications channel 89

disk subsystems 78
 components 79
 multiple 79
 disks
 bad blocks on disk 78
 DOS disks, accessing 85
 how blocks are numbered 78
 multiple operating systems on a disk 79
 partition types 79
 partitions, number per disk 79
 ramdisks 77
 used blocks 84
 viewed as block special files 78
 see also partitions
 Dosfsys filesystem manager 85
 dot (directory link) 71
 dot dot (directory link) 71
 driver processes
 Dev.con 94
 Dev.par 96
 Dev.ser 95
 drivers 5
 device drivers 92 – 93
 effects of adding new drivers 5
 network drivers 104
 dumper (utility) 45
 duplicating file descriptors 62

E

echoing characters 88
 edited input mode 89 – 90
 elevator seeking (Filesystem Manager) 75
 erroneous arithmetic operation
 signal generated 24
 Ethernet drivers/cards 104
 see also network
 event-reporting mechanisms
 notify with proxy 50

 notify with signal 50
 sleep until completion 50
 exec() (process creation primitive) 42
 executable files running with
 their own privileges 67
 executing processes 44
 see also processes
 extending files 69
 extensibility of QNX 4
 extents 69
 allocating new extents 69
 how files are extended 69
 where extents are stored 69

F

fault tolerance (network) 100, 107
 FIFO scheduling 34
 FIFOs 66, 75
 creating 75
 removing 75
 see also pipes
 file descriptions
 see open control blocks (OCBs)
 file descriptors (FDs)
 duplicating 62
 inheritance 62
 inheritance, preventing 63
 namespace 61 – 63
 open control blocks (OCBs) 61
 several FDs referring to the same OCB 62
 filenames
 defined 67
 dot (directory link) 71
 dot dot (directory link) 71
 longer than 16 characters 71
 see also links

files

- access date 66
- accessing 67
- accessing across the network 55
- block special files 66, 78 – 82
- character special files 66
- closing 71
- creation date 66
- date and time stamps 66
- defined 66
- directories 66 – 67
- DOS files, operating on 85
- executable files running with
 - their own privileges 67
- extending 69
- extents 69
- FIFOs 66, 75
- filenames 67
- links 70 – 71
- locating 67
- modification date 66
- multiple filenames 70 – 71
- network access 55
- opened by different processes 61
- opened twice by same process 61
- permissions 67
- pipes 66, 74
- pseudonyms 70 – 71
- recovering 78
- regular files 66 – 67
- remote files, accessing 55
- security 67
- symbolic links 66, 72 – 73
- times maintained for each file 66
- types of files implemented in QNX 66
- write date 66
- see also* directories; extents; filenames

filesystem

- accessing a filesystem on another node 58
- DOS filesystems 85
- guest filesystems 85
- implemented on block special files 67
- mounting 82
- navigating 67
- recovering 78
- robustness 77
- seek points 61
- unmounting 82
- see also* partitions

Filesystem Manager

- accessing several devices in parallel 76
- buffer cache 75
- client-driven priority 77
- Dosfsys filesystem manager 85
- elevator seeking 75
- filetypes managed by Fsys 66
- guest filesystems 85
- I/O requests from buffer cache while other
 - requests in progress 76
- I/O requests, managing 76
- multi-threading 76
- performance 75 – 76
- ramdisk 77
- responsibilities 66
- temporary files 77
- see also* files; filesystem; partitions;
 - read operations; write operations

flow control 88

flow of data between:

- applications and devices 88, 93
- Dev and its drivers 93

fork() (process creation primitive) 42

fullpath utility 60

G

global name server 49
 global names 49
 guest filesystems 85

H

hangups
 recognizing and acting on 88
 hard links
 see links
 hardware instructions, signal generated upon
 detection of invalid instruction 24
 hardware interrupts
 see interrupts
 HELD (process state) 46

I

I/O namespace 54
 alias prefixes 58
 current working directory 59
 default network root 56
 file descriptor namespace 61 – 63
 network root 55
 pathnames, resolving 54 – 60
 prefix aliases 58
 relative pathnames 59
 resolving pathnames 54 – 60
 symbolic links 72 – 73
 I/O servers
 interrupt-driven 96
 non-interrupt servers 96
 I/O services 54
 file-oriented 54
 images 84

inheritance
 see file descriptors
 see also processes
 inodes
 .inodes file 70
 where inode information is stored 70
 input
 edited input mode 89 – 90
 lines, editing 88, 90
 model 97
 raw input mode 91
 redirecting 74
 inserting
 characters 90
 interprocess communication (IPC) 5, 11
 across the network 29 – 31
 via messages 11 – 20
 via proxies 21
 via signals 22 – 28
 see also message passing; network; proxies;
 signals; virtual circuits; virtual processes
 interrupt handlers 38, 51 – 52
 placing received data into memory queue 97
 scheduling **Dev** to run 97
 timer interrupt handlers 52
 see also proxies
 interrupt latency 37
 interrupt-driven I/O servers 96
 see also non-interrupt I/O servers
 interrupts 37
 priorities 39
 stacked 39
 IOT instruction, signal generated 25
 IPC
 see interprocess communication

J

job control signals 24

K

kernel

- entered as result of kernel calls 4
- general responsibilities 3, 10
- interface with Network Manager 101 – 103
- scheduling processes 32 – 36
- size 3

kill (Shell built-in) 22

L

LAN

see network

latency

- interrupt latency 37
- scheduling latency 38

line editing 88

line-control discipline 88

links 70 – 71

- creating 71, 73
- directory 71
- limitations 70
- removing 71, 73
- symbolic 72 – 73
- see also* filenames

load balancing (network) 100, 106

loader block 83

loading processes 44

local area networking

see network

local names 49

locating

- files 67
- processes 48

logical networks 105

choosing 106

logical network IDs 105

see also network; Network Manager

logical node IDs 105

assigning 105

M

managers (system processes) 4

similarity to user-written processes 4

see also Device Manager; Filesystem Manager;
Network Manager; Pipe Manager;
Process Manager

memory parity error, signal generated 25

memory references, signal generated upon detection
of invalid reference 24

memory-image dump, generating when a process
is terminated by a signal 45

message passing 5, 11 – 20

advanced message-passing facilities 18
buffer space already allocated for a message,
using 19

conditional message reception 18

message data, where stored 16

messages containing variable amounts
of data 19

messages defined 11

messages, contents of 6, 11

messages, purpose of 11

multi-part messages 19

network-wide 29, 100 – 101, 103

order in which messages are received 17

primitives 11, 19

process synchronization 13

reading part of a message 19

remote messages, receiving 101, 103

remote messages, sending 101

reserved message codes 20

- signals and messages 28
 - synchronizing processes 13
 - writing part of a message 19
 - zero-length messages 16
 - zero-length replies 16
 - see also* interprocess communication; proxies; signals; virtual circuits
 - microkernel architecture 3 – 4
 - see also* kernel
 - modem
 - connected to a serial port 90
 - modification date (files) 66
 - mounting filesystems 82
 - see also* block special files
 - moving
 - cursor 90
 - multi-part messages 19
 - multi-sector writes 75
 - multi-threading (Filesystem Manager) 76
- ## N
- name server 49
 - see also* process name locator; process names
 - names
 - passing to other processes 57
 - see also* device names; filenames; process names
 - network
 - address conflicts when interconnecting networks 104
 - choosing 106 – 107
 - default network root 56
 - drivers 104
 - drivers, responsibilities 104
 - fault tolerance 100, 107
 - flexibility 7
 - as homogeneous set of resources 6 – 8
 - identifiers 104 – 105
 - integrity 31
 - interconnected networks 105
 - IPC, network-wide 29 – 31
 - load balancing 100, 106
 - logical network IDs 105
 - logical networks 105
 - logical networks, choosing 106
 - logical node IDs 105
 - mapping between logical and physical node IDs 105
 - message passing 100 – 101, 103
 - node IDs 104 – 105
 - nodes leaving the network 31
 - non-networked machines 101
 - pathnames 55, 59
 - pathnames, displaying 60
 - physical node IDs 104
 - polling 105
 - remote files, accessing 55
 - root 55
 - throughput improved via
 - load balancing 106
 - transparency 6 – 7, 29, 100
 - virtual processes 29
 - Network Manager 99 – 107
 - drivers 104
 - drivers, responsibilities 104
 - independence 100
 - interface with kernel 101 – 103
 - mapping between logical and physical node IDs 105
 - responsibilities 100
 - starting and stopping 100
 - virtual circuits 29
 - see also* network
 - nodes
 - logical node IDs 105
 - physical node IDs 104
 - see also* network; Network Manager
 - non-interrupt I/O servers 96

- notify with proxy
 - (event-reporting mechanism) 50
- notify with signal
 - (event-reporting mechanism) 50

O

- on** utility 57
- open control blocks (OCBs) 61
- open resources
 - active information contained in OCBs 61
- orphan processes 44
 - see also* processes
- OS
 - images 84
 - partitions 79, 81
- output
 - redirecting 74
- output characters, translation of 88
- overwriting characters 90

P

- parallel devices 96
 - throughput, maximizing 96
- parallel ports
 - determining if port is available 96
 - installing 96
- parent directory 71
- parent processes
 - effects of death on children 44
 - see also* processes
- parity error, signal generated 25
- partitions 79
 - bitmap 84
 - creating, modifying, or deleting 80
 - DOS partitions 85
 - flagging as bootable 80

- key components 83 – 84
- loader block 83
- primary partitions, number per disk 79
- root block 84
- root directory, special characteristics 84
- types 79
 - see also* filesystem; Filesystem Manager
- passing names to other processes 57
- pathname prefix tree 54
- pathname space
 - location 54
 - regions of authority 54
 - see also* I/O namespace
- pathnames 67
 - converting relative pathnames
 - to network pathnames 59
 - network pathnames 55
 - network pathnames, displaying 60
 - resolving 54 – 60
- pending signals 23
- performance
 - device subsystem 97
 - fast interactive response via
 - adaptive scheduling 35
 - Filesystem Manager 75 – 76
 - network throughput 106
 - realtime 37 – 40
 - see also* throughput
- permissions 67
- physical node IDs 104
- Pipe Manager 74
- pipes 66, 74
 - creating 74
 - effect of writing on a pipe
 - with no readers 24
 - see also* FIFOs
- polling the network 105
- prefix** (utility) 55
- prefix aliases 58

- prefix root
 - see* default network root
- prefix tree 54
 - displaying 55
 - managed as a list of prefixes 55
- prefixes 54
- priorities 32
 - determining 33
 - setting 33
- process control environments 7
- process creation primitives 42
- process inheritance 43
- Process Manager
 - general responsibilities 42
- process name locator 48 – 49
- process names
 - attaching 49
 - global names 48
 - local names 48
 - locating processes by name 49
 - reducing potential name conflicts 49
- process states
 - DEAD 46
 - determining 48
 - HELD 46
 - READY 46
 - RECEIVE-blocked 13, 46
 - REPLY-blocked 13, 46
 - SEND-blocked 13, 46
 - SIGNAL-blocked 28, 46
 - WAIT-blocked 46
- processes
 - catching signals 26
 - child processes becoming zombies 45
 - child processes inheriting environment 43
 - child processes running concurrently
 - with parents 44
 - child processes, creating 42
 - cooperating 48
 - cooperating via messages 11
 - cooperating via pipes and FIFOs 74
 - creating 44
 - creating on any node 42
 - creation primitives 42
 - death of parent process,
 - effects on children 44
 - death of session leader, signal generated 24
 - executing 44
 - executing on remote nodes 56
 - inheritance 43
 - life cycle 44 – 45
 - loading 44
 - locating 48
 - locating by name 49
 - names (symbolic) 48
 - opening a file 54
 - opening the same file twice 61
 - orphan processes 44
 - parent processes, death of 44
 - passing names to 57
 - primitives 42
 - priorities, determining 33
 - priorities, setting 33
 - process states 46 – 47
 - remote execution 56
 - scheduling 32 – 36
 - scheduling (adaptive) 35
 - scheduling (client-driven) 36
 - scheduling (FIFO) 34
 - scheduling (round-robin) 35
 - signal generated upon death of
 - session leader 24
 - signal generated when child process
 - terminates 24
 - signals, catching 26
 - signals, effects of 23
 - signals, protection against 25
 - signals, receiving 23
 - states 46 – 47
 - states, determining 48

- symbolic names 48
- synchronizing via message passing 13
- system processes 4
- termination 44
- termination by a signal 45
- termination, effect on virtual circuits 29
- termination, effects of 44
- virtual processes 29
- zombie processes 45 – 46

programs

- see* processes

proxies

- creating 21
- processes that can use a proxy 21
- purpose of 11, 21
- used by interrupt handlers 21

Q

QNX

- console 94
- defined 2
- extensibility 4
- flexibility 2
- fully distributed OS 100
- microkernel architecture 3 – 4
- network as homogeneous set
 - of resources 6 – 8
- network flexibility 7
- network transparency 7
- realtime applications, suitability for 2
- single-computer model 7

R

- ramdisk (Filesystem Manager) 77
- raw device, reading from 91
- raw input mode 91

- raw volumes 78 – 82
 - see also* disks; partitions
- read operations (**Dev**)
 - console screen 95
 - non-blocking reads 89
 - non-blocking writes 89
 - parameters defining when read request
 - to be satisfied 91
 - raw device 91
 - timed read operations 89
- read operations (**Fsys**)
 - priority over write operations 76
 - synchronous 76
- reading directory entries 67 – 68
- READY (process state) 46
- realtime performance 2, 37 – 40
 - interrupt latency and 37
 - scheduling latency and 38
 - stacked interrupts and 39
- rebooting
 - signal generated 25
- Receive()* (message-passing primitive) 11, 100
 - using 15
- RECEIVE-blocked (process state) 13, 46
- recovering your filesystem 78
- redirecting
 - input 74
 - output 74
- regular files 66 – 67
 - extents 69
- relative pathnames
 - converting to network pathnames 59
- remote execution 56
- remote files, accessing 55
- removing timers 51
- Reply()* (message-passing primitive) 11, 16, 100
 - to a remote node 101
 - using 15
- REPLY-blocked (process state) 13, 46
- reply-driven messaging 16

- resolving pathnames 54 – 60
 - see also* pathnames
- resource managers
 - see* managers
- resources
 - accessing on other machines 6
 - open 61
- restoring your filesystem 78
- root
 - default network root 56, 59
 - network root 55
 - super root 55
- root block 84
- root directory
 - special characteristics 84
- round-robin scheduling (processes) 35

S

- scheduling latency 38
- scheduling processes 32 – 36
 - adaptive scheduling 35
 - client-driven priority scheduling 36
 - FIFO scheduling 34
 - process priorities 32
 - round-robin scheduling 35
 - scheduling method, determining 34
 - scheduling method, setting 34
- security (files)
 - see* files
- seek points 61 – 62
- Send()* (message-passing primitive) 11, 100
 - to a remote node 101
 - using 14
- SEND-blocked (process state) 13, 46
- serial channels
 - managing 95
- serial devices 95
- serial ports
 - installing 95
 - seeing which ports are available 95
- servers 16, 28, 36
- services, handled by QNX processes 4
- session leader
 - signal generated upon death 24
- setgid 67
- setuid 67
- Shell
 - see User's Guide and Utilities Reference*
- shutdown** (utility)
 - signal generated 25
- signal handlers 23, 26 – 27
- signal handling functions 26
- SIGNAL-blocked (process state) 46
- signals
 - blocking delivery 27
 - catching 26
 - default actions 23 – 24
 - defining signal processing 26
 - delivery of 22 – 23
 - effect when process is SIGNAL-blocked 29
 - effects when process is REPLY-blocked 28
 - functions safe to use within signal handlers 26
 - generating 22
 - ignoring signals, effects of 23
 - messages and signals 28
 - order of delivery 23
 - pending 23
 - processes protecting themselves
 - against signals 25
 - processes setting signals on themselves 22
 - processes, effects of signals on 23
 - purpose of 11, 22
 - receiving 23
 - summary of signals 24
- single-computer model 7
- slay** (utility) 22

- sleep until completion
 - (event-reporting mechanism) 50
- software interrupts
 - see* signals
- spawn()* (process creation primitive) 42
- stacked interrupts 39
- states of processes
 - see* process states
- super root 55
- symbolic links 66, 72 – 73
 - cd* command and 60
 - creating 73
 - querying 73
 - removing 73
 - see also* links
- symbolic names (of processes) 48
 - see also* process names
- synchronizing processes via message passing 13
- system consoles 94
- system managers 4
 - similarity to user-written processes 4
 - see also* Device Manager; Filesystem Manager; Network Manager; Pipe Manager; Process Manager

T

- temporary files (Filesystem Manager) 77
- terminal control structure (*termios*) 88
- terminal devices
 - accessing 88
 - connected via a serial channel 90
 - device services 88
 - hangup on controlling terminal,
 - signal generated 24
 - located in I/O namespace 88
 - names 88

- representing bidirectional stream of bytes
 - to processes 88
 - see also* devices
- terminating processes 44
 - see also* processes
- termios* control structure
 - ICANON bit 89 – 91
- termios* parameters
 - functionality controlled by 88
- throughput (network) 106
- throughput (parallel devices) 96
- time management 49
 - see also* timers
- time stamps 66
- timers 49 – 50
 - advanced timing 50
 - arming 50
 - creating 50
 - reading 51
 - removing 51
 - setting resolution of 51
 - simple timing 49
 - timer interrupt handlers 52
- Token Ring 104
 - see also* network
- transparency of QNX networks 7, 100
 - see also* network
- TZ (environment variable) 49

U

- unmounting filesystems 82

V

- virtual circuits (VCs) 29
 - automatic resizing of buffers 29
 - automatic resizing to accommodate large messages 29
 - created on behalf of the application 31
 - effect of terminating process on 29
 - integrity check 31
 - setting up 29
 - sharing 29
 - terminating 31
- virtual consoles
 - defined 94
 - see also* consoles
- virtual processes 29

W

- WAIT-blocked (process state) 46
- window change, signal generated 25
- working directory
 - see* current working directory
 - see also* directories
- write date (files) 66
- write operations (**Dev**)
 - writing to console screen 95
- write operations (**Fsys**)
 - asynchronous 76
 - modifying write behavior file-by-file 76
 - multi-sector writes 75
 - priority of read operations 76
 - write requests, method of writing 75

Z

- zero-length messages 16
- zero-length replies 16
- zombie processes 46
 - see also* processes

