

QNX4.0

User's Guide



QNX[®] 4.0 Operating System

User's Guide

Quantum Software Systems Ltd.

175 Terrence Matthews Crescent

Kanata, Ontario K2M 1W8

Canada

(613) 591-0931

© Quantum Software Systems Ltd. 1991

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 Quantum 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.

This edition: April, 1991

QNX is a registered trademark of Quantum Software Systems Ltd.

UNIX is a registered trademark of AT&T.

Cover art by Robert Tinney.

Printed in Canada.

Acknowledgments

Several members of Quantum Software Systems Ltd. have contributed to the design and implementation of QNX 4.0. This product embodies more than a decade of research, development, and field-tested experience. Veteran QNX users will recognize the performance advantages and comfortable feel of their familiar OS, while new users can expect a “standard” operating system with a difference. 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 4.0:

Software Development:

Luc N. Bazinet
Gordon Bell
Andrew C. Boyd
Brian Campbell
Grant D. Courville
Dan Dodge
Andrew Edgar
William A. Flowers
Dan Frank
Jeffrey G. George
Dan Hildebrand
Eric Johnson
Debbie Kennedy
Terry Laughlin
Steve McPolin
Rob Oakley
Martin Walter

Documentation:

Kevin Didsbury
Paul N. Leroux
Paul McManus
John M. Ostrander



About this Guide

What will I find in this guide?

The QNX operating system *User's Guide* accompanies the QNX operating system (release 4.0) and is intended for both application developers and end-users.



We assume that QNX is already installed and running on your machine. If not, you—or your system administrator—should refer to the *Administrator's Guide* for installation and configuration.

The *User's Guide* contains two main parts:

- **System Overview**—a detailed look at the structure and functions of QNX
- **Using the QNX Shell**—a quick tour through some basic operations as well as a topical reference

Conventions used in this guide

Technical terms

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions.

Reference	Example
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}

How keys are displayed in print

When this guide instructs you to press keys on your keyboard, the keys you're to type appear between angle brackets. For example, if a passage instructs you to:

press <Enter>

you press the key labeled Enter.

Key chords

When this guide instructs you to press more than one key at the same time, the keys are separated by a hyphen (-). For example, if you see:

press <Ctrl>-<N>

you would:

- 1) press <Ctrl> and hold it down
- 2) press <N>
- 3) quickly release both keys.

Entering text

When this guide instructs you to enter text from your keyboard, the text you should type appears in a **constant-width typeface**. For example, if you see:

Type **pwd**

you would type:

pwd

Important notes

Important text is indicated by a pointing hand (☞). For example:



If you want to move a file from another user's directory, you must first have the appropriate permissions on that directory.

A word about POSIX

QNX 4.0 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 now proliferating on the market. Sanctioned by several internationally recognized standards bodies and supported by numerous industry participants, the POSIX standard serves as a verifiable means to port programs across various environments.

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

Contents

Part I System Overview

Chapter 1	The Philosophy of QNX	3
	What is QNX?	4
	QNX's microkernel architecture	5
	A true kernel	5
	System processes	6
	Interprocess communication	7
	A message-passing operating system	7
	QNX as a network	8
	The single-computer model	9
	Flexible networking	9
Chapter 2	The Microkernel	11
	Introduction	12
	Interprocess communication (IPC)	13
	IPC via messages	13
	Message-passing primitives	13
	Process synchronization	15
	Blocked states	15
	Using <i>Send()</i> , <i>Receive()</i> , and <i>Reply()</i>	16
	Reply-driven messaging	18
	Additional points to remember	18

The Microkernel (cont'd)

- Advanced facilities 20
- Reserved message codes 22
- IPC via proxies 23
- IPC via signals 24
 - Generating signals 24
 - Receiving signals 25
 - Default actions 25
 - Summary of signals 26
 - Defining signal processing 28
 - Catching signals 28
 - Safe functions for signal handlers 28
 - Blocking signals 29
 - Signals and messages 30
- IPC across the network 31
 - Virtual circuits 31
 - Virtual processes 31
 - Terminating virtual circuits 33
- Process scheduling 35
 - When scheduling decisions are made 35
 - Process priorities 35
 - Scheduling methods 36
- A word about realtime performance 40
 - Interrupt latency 40
 - Scheduling latency 41
 - Stacked interrupts 42

Chapter 3 The Process Manager 45

- Introduction 46
 - Process Manager responsibilities 46
 - Process creation primitives 46
 - Process inheritance 47

The life cycle of a process	48
Creation	48
Loading	48
Execution	48
Termination	48
Process states	50
Determining process states	52
Process symbolic names	52
Timers	53
Time management	53
Simple timing facilities	53
Advanced timing facilities	54
Interrupt handlers	55
Timer interrupt handlers	56

Chapter 4 I/O Namespace 57

Introduction	58
I/O Namespace	58
Prefixes and regions of authority	58
Resolving pathnames	58
Network root	59
Relative pathnames	60
Prefix aliases	60
Private prefixes	62
File descriptor namespace	63
Open control blocks	64

Chapter 5 The Filesystem Manager 67

Introduction	68
What is a file?	68
Date and time stamps	68
Regular files and directories	69
Regular files	69
Directories	69
Extents	72

The Filesystem Manager (cont'd)

- Pipes and FIFOs 74
 - Pipes 74
 - FIFOs 74
- Filesystem Manager performance 75
 - Elevator seeking 75
 - Buffer cache 75
 - Multi-threading 76
 - Client-driven priority 76
 - Temporary files 76
 - Ramdisks 76
- Filesystem robustness 77
 - Filesystem recovery 77
- Raw volumes 78
 - Disks and disk subsystems 78
 - OS partitions 78
 - Defining block special files 80
 - Mounting a filesystem 81
- Key components of a QNX partition 82
 - Loader block 82
 - Root block 83
 - Bitmap 83
 - Root directory 83

Chapter 6 The Device Manager 85

- Introduction 86
- Device services 86
- Edited input mode 87
- Raw input mode 89
- Device drivers 90
 - Device control 92
- The QNX console 92
 - Switching virtual consoles 93
 - Changing the screen size 94
 - Console-specific functions 94

Serial devices	95
Parallel devices	96
Device subsystem performance	97
International keyboards	97

Part II **Using the QNX Shell**

Chapter 1 **Using the Keyboard 101**

Introduction	102
Entering line-oriented input	102
Line-editing keys	102
Maximum length of an input line	103
Entering long input lines	103
Recalling commands	103
Switching virtual consoles	104
Using multiple consoles	104
Changing the screen size	105
Suspending and resuming output	106
Rebooting	106
The keyboard at a glance	107

Chapter 2 **Command Basics 109**

Understanding command syntax	110
Syntax structure	110
Key symbols	111
Grouping options	112
Arguments containing spaces	112
Displaying online usage messages	113
Redirecting input and output	113
Using wildcards	114
Entering multiple commands	115

Chapter 3 Working in the QNX Filesystem 117

- Introduction 118
- Filenames 118
 - Portable filenames 118
 - Filename suffixes 118
- Pathnames 119
 - Network pathnames 119
 - Current working directory 119
 - Home directory 120
- A typical QNX filesystem 120
- Commonly used commands 122

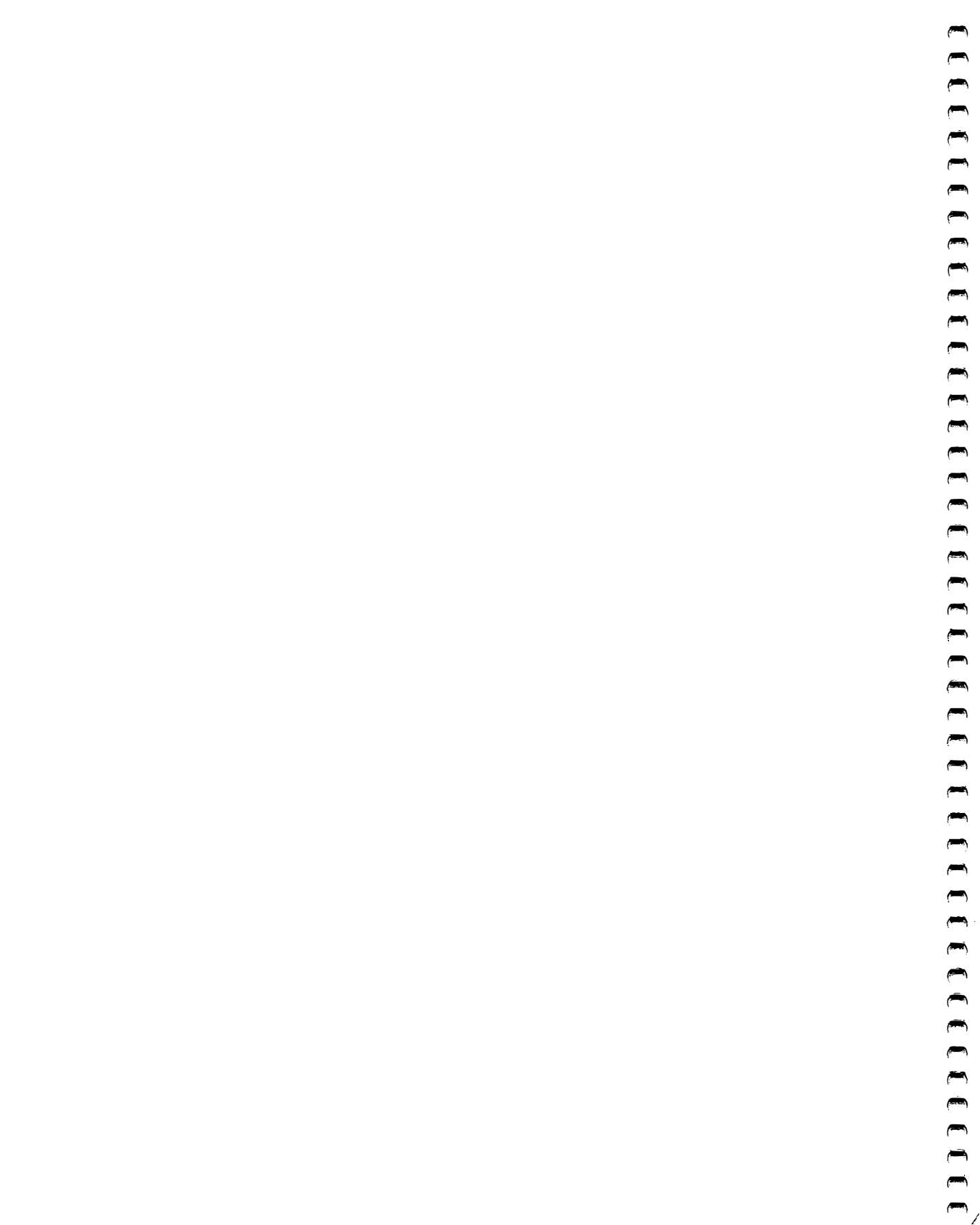
Chapter 4 Shell Topical Reference 123

- About the Shell 125
- Command lists 126
 - Sequential lists 126
 - Background lists 126
 - AND/OR lists 126
 - Grouping command lists 127
- Command substitution 127
- Environment variables 127
- Expansion 128
 - Arithmetic expansion 128
 - Filename expansion 129
 - Parameter expansion 130
 - Tilde expansion 132
- Invocation modes 133
- Pipes 134
- Quoting 135
 - Escape character (backslash) 135
 - Single quotes 135
 - Double quotes 136
 - Shell prompts 136
- Redirection 137

Remote execution	139
Reserved words	140
Shell scripts	140
Signals	141
Terminology	142
Using Shell commands	143
Control Constructs	144
Regular built-in utilities	146
Special built-in commands	150

Appendix A	The QNX Terminal	157
	Standard console character set	160
	QNX keyboard codes	162
	Composed character input sequences	164
	Terminal escape sequences	165

Index	169
--------------	------------



Part I

System Overview

In this Part...

The Philosophy of QNX

The Microkernel

The Process Manager

I/O Namespace

The Filesystem Manager

The Device Manager



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.

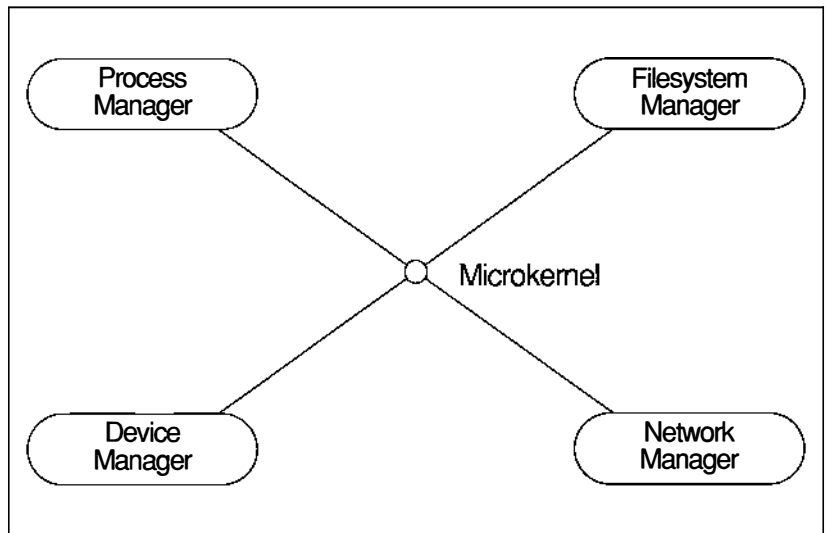
QNX is also remarkably flexible. Developers can easily customize the operating system to meet the needs of their application. From a “bare-bones” configuration of a kernel with a few small modules to a full-blown network-wide system equipped to serve hundreds of users, QNX lets you set up your system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

System processes

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.

All QNX services, except for 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 (or 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 clean room.

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

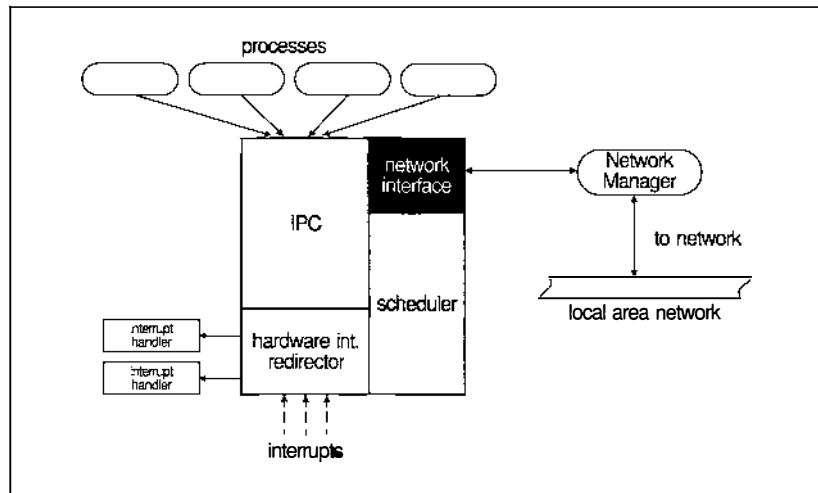
Process scheduling

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 be the next to be given control of the CPU
- **first-level interrupt handling**—all hardware interrupts and faults are first routed through the Kernel, then passed on to the appropriate system manager



The QNX Microkernel.

Interprocess communication (IPC)

QNX supports three 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 of the message and for the recipient, but for no one else.

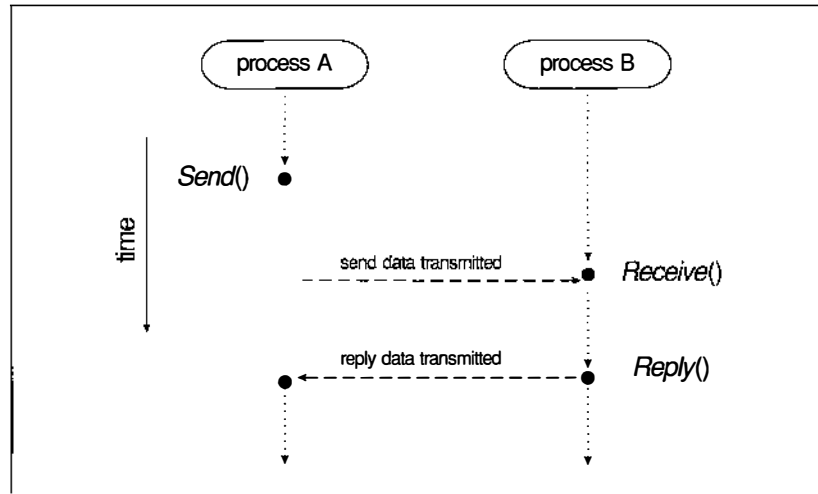
Message-passing primitives

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.



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, it 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.



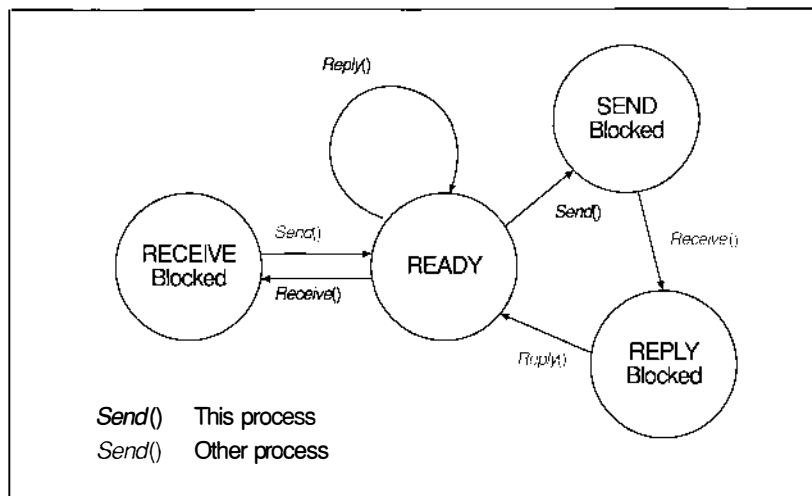
For details on how QNX schedules processes, see “Process scheduling” in this chapter.

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, The Process Manager.

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, smsg, rmsg, smsg_len, rmsg_len );
```

The *Send()* call contains these arguments:

- pid* the process ID of the process that is to receive the message (i.e. *Process B*); a *pid* is the identifier by which the process is known to the operating system and to other processes
- smsg* the message buffer (i.e. the message to be sent)
- rmsg* the reply buffer (will contain the reply from *Process B*)
- smsg_len* the length of the message being sent

rmsg_len 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:

pid 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

msg the buffer where the message will be received

msg_len 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 aren't the same 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* might reply to *Process A* by issuing a *Reply()* function call:

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

The *Reply()* call contains these arguments:

pid the process ID of the process to which the reply is directed (i.e. *Process A*)

reply the reply buffer

reply_len 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 aren't the same 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, it's 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

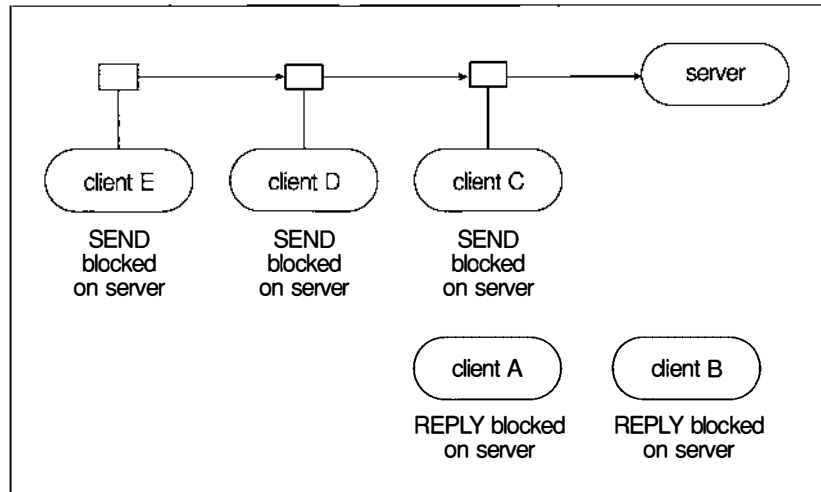
Here are some more things to keep in mind about message passing:

- 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.

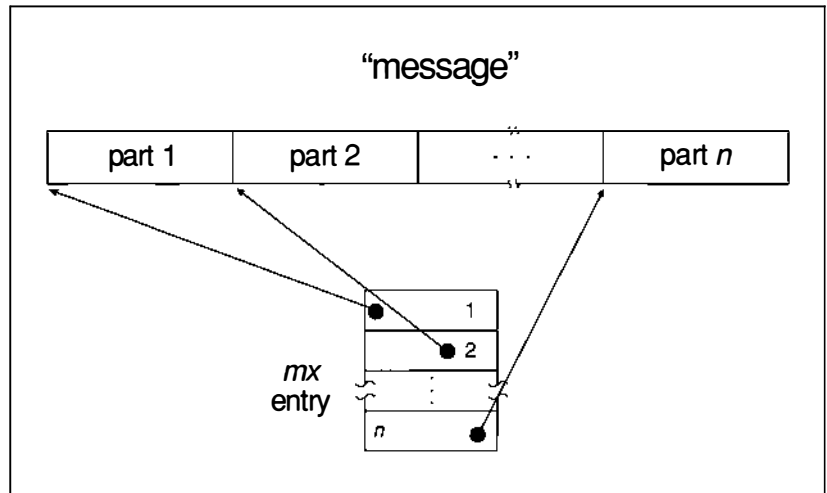
Advanced facilities

QNX also provides these advanced message-passing facilities:

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

Multi-part messages

Up to now, messages have been discussed as single packets of bytes. But to reduce copying into buffers—and thereby improve system efficiency—messages often consist of two or more discrete components. For example, a message can have a fixed-length header followed by a variable amount of data. To efficiently support this need, QNX allows a message to be constructed from two or more separate message buffers.



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

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

- *`Creceivemx()`*
- *`Readmsgmx()`*
- *`Receivemx()`*
- *`Replymx()`*
- *`Sendmx()`*
- *`Writemsgmx()`*

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.

**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

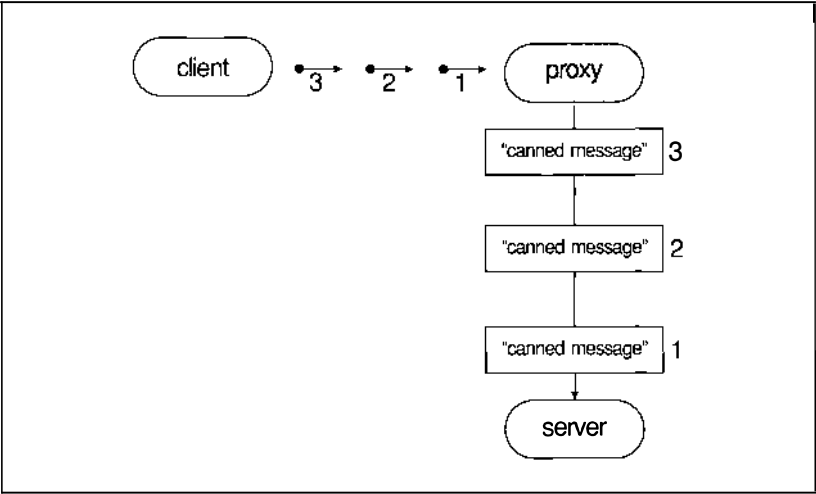
A *proxy* is a form of non-blocking message that is especially suited for event notification where the sending process doesn't need to interact with the recipient. The only function of a proxy is to send a fixed message to a specific process that owns the proxy. Like messages, proxies work across the 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 acknowledgement 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:	you use the:
generate a signal from the shell	<code>kill</code> or <code>slay</code> utilities
generate a signal from within a process	<code>kill()</code> or <code>raise()</code> 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 one signal, SIGKILL, 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 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.

Default actions

A number of possible *default actions* are defined for signals. These include:

- ignore the signal
- cause the process to pause
- terminate the process (the process exits abnormally)
- continue a paused process; ignore the signal if the process isn't paused
- treat as an unsupported signal

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, such as division by zero or an operation resulting in overflow
SIGHUP	yes	"	Death of controlling process, 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 or paused
SIGCONT	no	Continue a paused process; ignore the signal if the process isn't paused	Continue if paused
SIGSTOP*	no	Pause the process	Pause process signal

Signal	Catch or ignore	Default action	Description
SIGTSTP*	no	Treat as unsupported 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	Terminate the process	Generated when a significant and requested event occurs in the Device Manager
SIGPWR	yes	Terminate the process	Soft boot requested via <Ctrl>-<Alt>-<Shift>- or shutdown utility
historical UNIX signals			
SIGIOT†	yes	—	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
SIGTRAP	yes	Terminate the process	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). A signal handler must be re-entrant with respect to the host process.

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.

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 defines a set of functions that let you block delivery of signals. A signal that is blocked remains pending; once it's unblocked, it's 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 recatch the signal. 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 *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*, which are paths the network manager provides to transmit messages, proxies, and signals across the network.

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

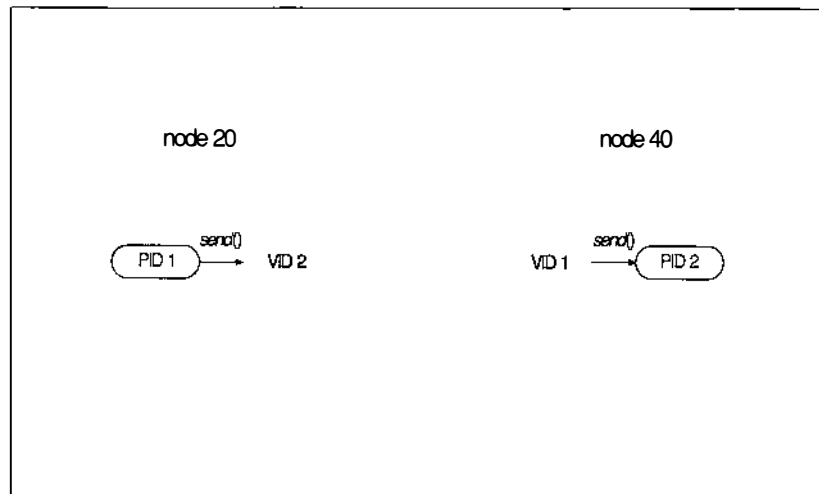
- When a virtual circuit is created, it's given the ability to handle messages up to a specified size; this means you can preallocate resources to handle the message. Nevertheless, if you need to send a message larger than the maximum specified size, the virtual circuit is automatically resized to accommodate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

Virtual processes

A sending process is responsible for setting up the virtual circuit 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 virtual circuit, 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 virtual processes.

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, 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*
- remote *vid*

You probably won't come into direct contact with virtual circuits very often. For example, when an application wants to access an I/O resource across the network, a virtual circuit 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 virtual circuit. Again, when an application establishes the location of a server with the *qnx_name_locate()* C function, a virtual circuit is automatically created on behalf of the application. To the application, the virtual circuit 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 virtual circuit 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 virtual circuit. 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 virtual circuits on its node. It does this as follows:

- 1) Each time a successful transmission takes place on a virtual circuit, a time stamp associated with the virtual circuit is updated to indicate the time of last activity.
- 2) At installation-defined intervals, the Process Manager looks at each virtual circuit. 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 virtual circuit 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 virtual circuit is dismantled; any process blocked on the virtual circuit 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

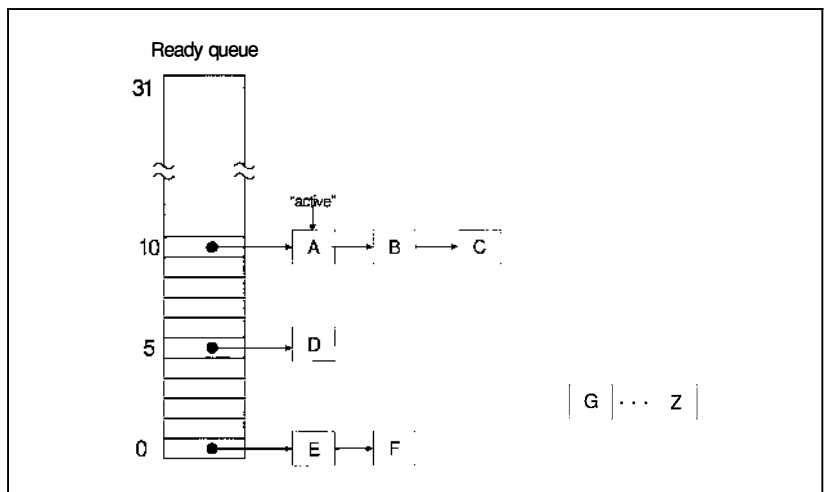
When scheduling decisions are made

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 launched from the Shell.

If you want to:	you 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

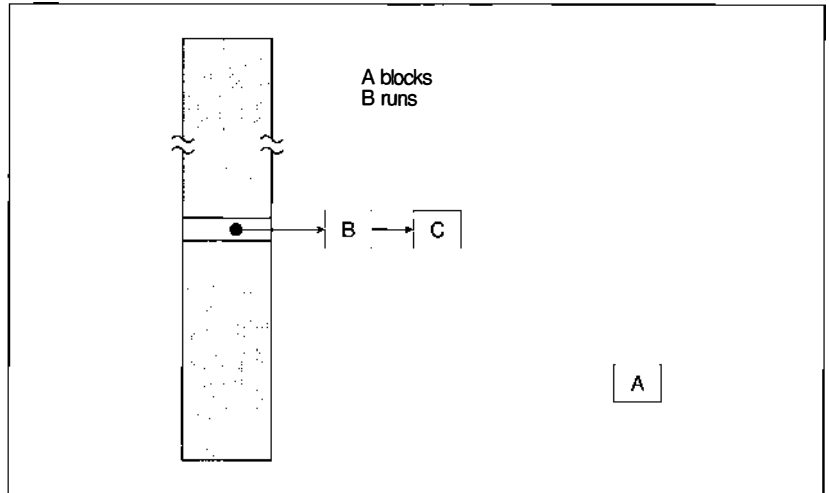
To meet the needs of various applications, QNX provides three scheduling methods:

- 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:	you use the:
determine the scheduling method for a process	<i>getscheduler()</i> C function
set the scheduling method for a process	<i>setscheduler()</i> 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.



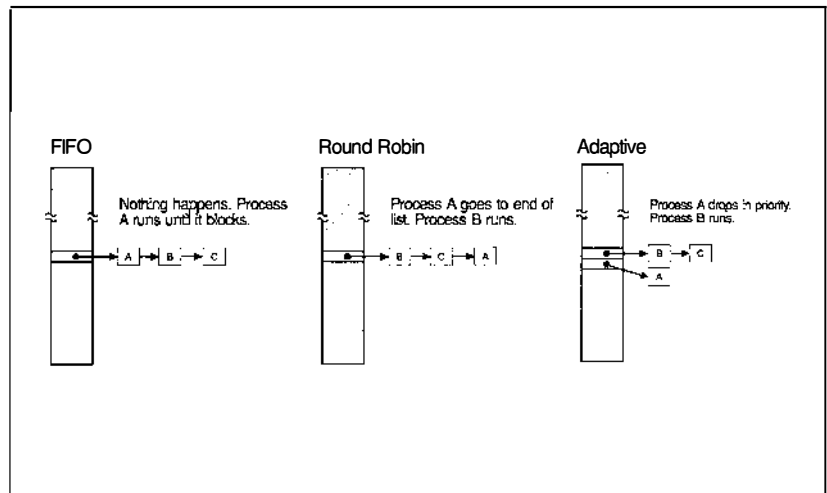
Apart from time slicing, round-robin scheduling is identical to FIFO scheduling.

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—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 for general use 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.



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 it 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

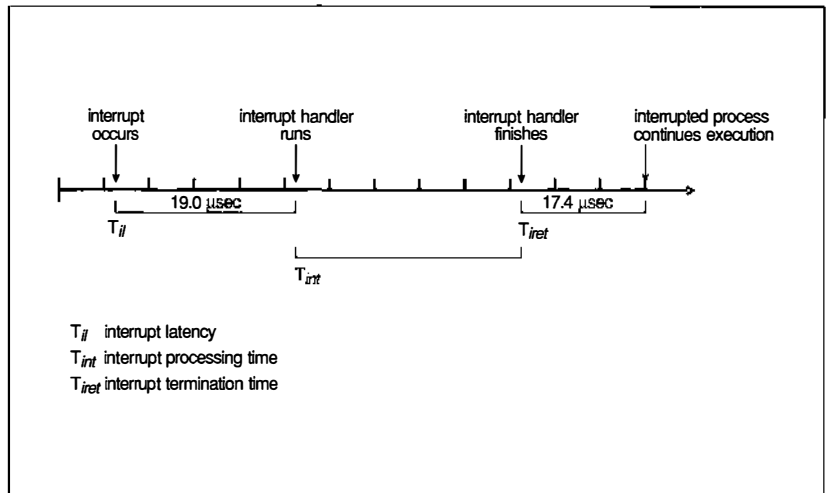
No matter how much we wish it were so, computers are not infinitely fast. In a realtime system, it's absolutely crucial that CPU cycles aren't unnecessarily spent. It's also crucial that you minimize the time it takes from the occurrence of an external event to the actual execution of code within the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



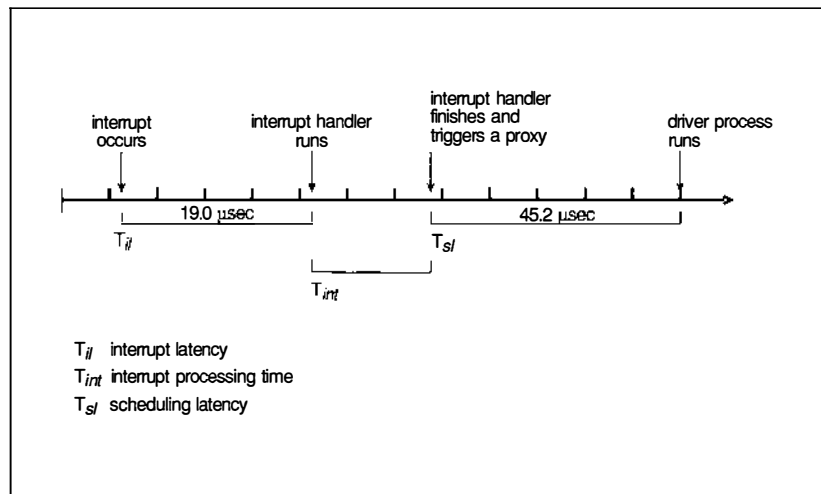
Interrupt handler simply terminates. Times are for a 20Mhz 386 processor in protected mode.

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.



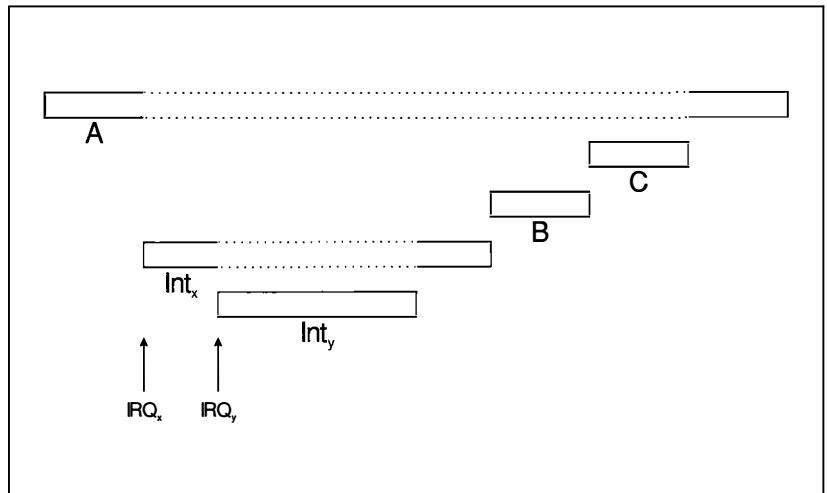
Interrupt handler terminates and triggers a proxy. Times are for a 20Mhz 386 processor in protected mode.

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.

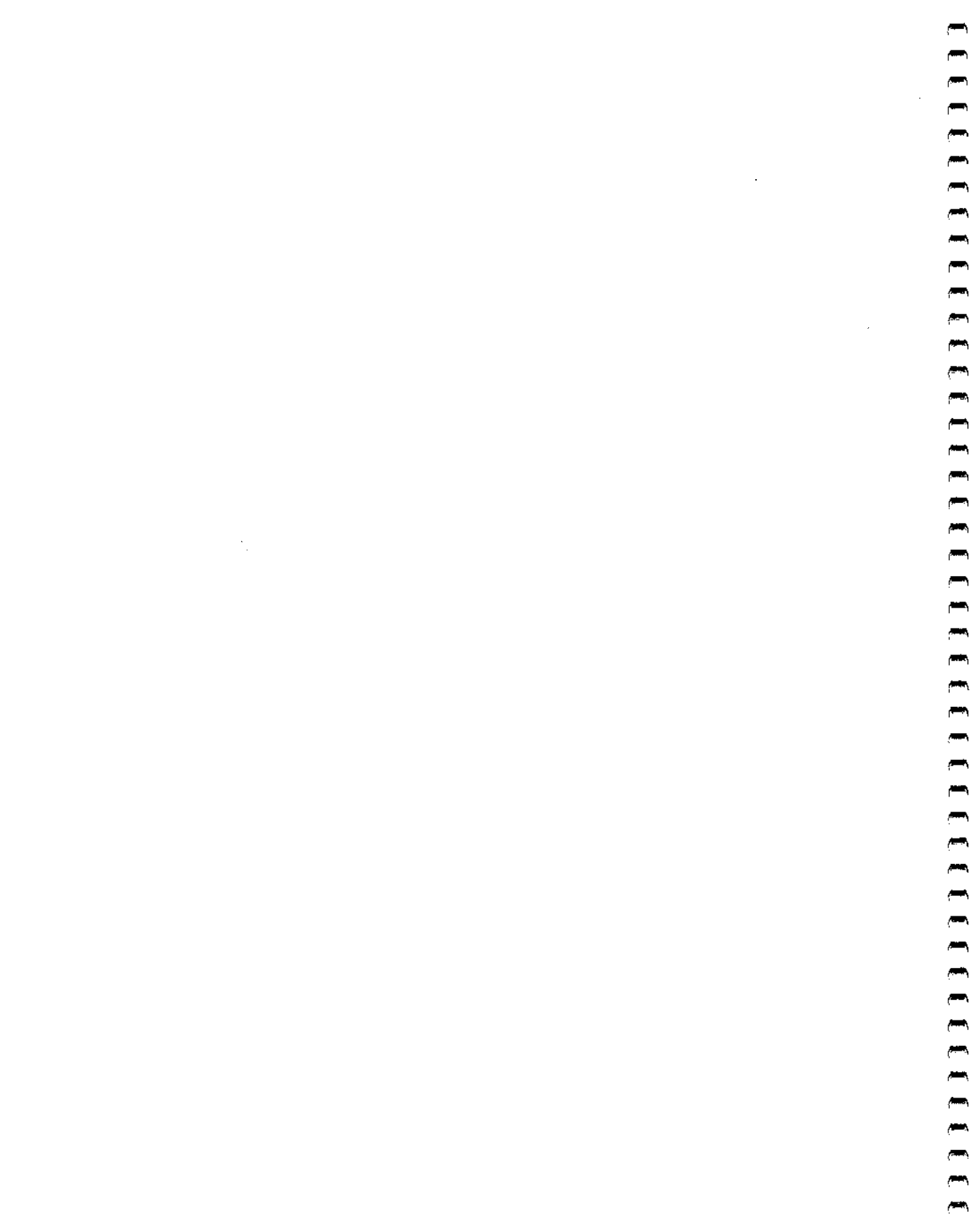
Stacked interrupts

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, 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 processes 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 *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 is no return from a successful *exec()*, because the calling process image is overlaid by the new process image. It is 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 in 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* cause automatic 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

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.

(You can optionally suppress this behavior; see *qnx_spawn()* and *qnx_spawn_options()* in the *WATCOM C Library Reference*.)

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.

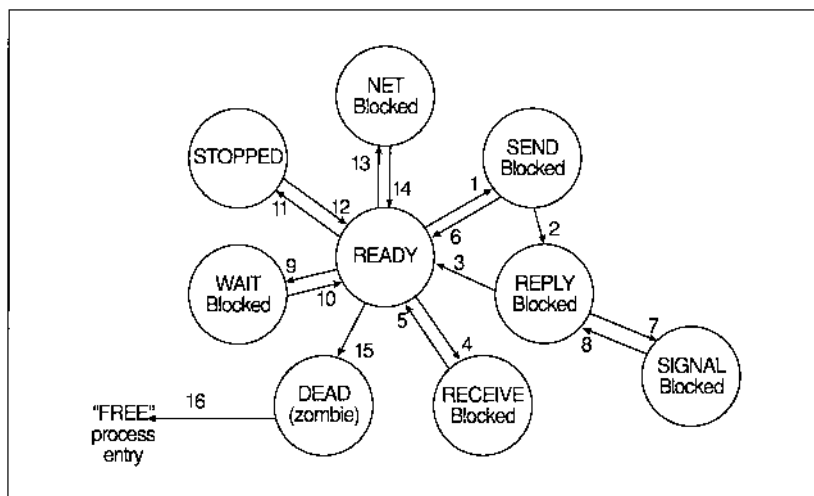
Process states

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
- **STOPPED**—the process has received a SIGSTOP signal. Until it's removed from the stopped state, a stopped process isn't eligible to use the CPU; the only way to remove it from the stopped state is to either deliver a SIGCONT signal or terminate the process.
- **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 Network control operation requested via `qnx_vc_attach()`.
- 14 Network transaction complete.
- 15 Process dies.
- 16 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 remains continuously 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:

<code>qnx/fsys</code>	local name
<code>company/xyz</code>	local name
<code>/company/xyz</code>	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

All 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 *Administrator's Guide*).

Simple timing facilities

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

Advanced timing facilities

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.

Creating timers

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 the *Administrator's Guide*).

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 when the timer is to go off
- **relative**—the time relative to the current clock value when the timer is to go off

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:
 - if the timer is active, the new interval is *added* to the current time interval
 - if the timer has expired, the new interval *replaces* the current time interval

To arm timers with:	you 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 ticksize utility or the <i>qnx_timerperiod()</i> C function. You can adjust the resolution from 500 μsec 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:	you 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. It will be invoked on each timer interrupt. To set the period, you can use the `ticksiz` 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

Prefix aliases

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 grafted on top of 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

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:
<code>/dev/con1</code>	<code>/dev</code>	Dev
<code>/dev/hd0</code>	<code>/dev/hd0</code>	Fsys
<code>/usr/dtdodge/x</code>	<code>/</code>	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, 0: /dev=5, 0: /=3, 1`

You can display the prefix tree using the **prefix** utility.

Network root

As noted earlier, each computer (node) maintains its own prefix tree. Every pathname that starts with a single slash is applied against the tree on the current node. 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 pathname space. For example, in a typical QNX network the following paths would map as follows:

<code>/dev/ser1</code>	local serial port
<code>//10/dev/ser1</code>	serial port on node 10
<code>//0/dev/ser1</code>	local serial port
<code>//20/usr/dtdodge/x</code>	file on node 20

Note that `//0` always refers to the local node. If you enter a pathname that starts with a single slash, the system assumes that it starts with the `//0` pathname.

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 this string to the path. For example, if your current working directory is:

```
//10/usr/dtdodge
```

then a request for the pathname “**x**” would result in a request for:

```
//10/usr/dtdodge/x
```

Both the `cd` Shell command and the `chdir()` C function check that the destination directory exists, and then modify this string. The current working directory will be inherited by child processes.

Prefix aliases

In addition to prefixes that map to an I/O resource manager, there is a second form of prefix known as an *alias prefix*. This type of prefix is a simple string substitution for a matched prefix. An alias prefix is of the form:

prefix=replacement-string

For example, assume that you’re running on a machine that doesn’t have a local filesystem (so there is no process to take over “/”). However, there is 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/x` will be replaced with the following:

```
//10/usr/dtdodge/x
```

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, 0 : /=//10/
```

Note that requests for pathnames under `/dev` will still be routed to the local character device manager, while requests other than `/dev` will be routed to the remote filesystem.

Aliases can also be used 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 directly named as:

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

Private prefixes

So far we've discussed the "system" prefix tree present on each node in a QNX network. Each process can also have a private prefix tree local to itself.

When a pathname is resolved, a search is made of both the private prefix tree and the system tree. If a match occurs in both trees, the one with the longest match will be selected. In the event of a tie, the private prefix tree will take priority.

For example, let's assume the system tree contains the prefix:

```
/=3,1  local filesystem
```

If we add the following prefix to the local prefix tree:

```
/=//20/
```

then that process has effectively moved its root to the root of node 20. Only that one process will be affected (as well as any children it subsequently creates).

The private tree can contain only alias prefixes, while the system tree can contain resource manager prefixes or aliases. The local tree lets you create your own private mappings or override those in the system.

The private tree also plays an important role when a process is spawned remotely on another node. For example, suppose a process without a private prefix tree decided to spawn the following command:

```
//20 wc /tmp/x
```

This would run **wc** on node 20, and resolve the pathname **/tmp/x** using node 20's system prefix tree, which may be different from the current node's system prefix tree. One way to change this behavior is to explicitly name the prefix tree you wish searched:

```
//20 wc //10/tmp/x      (if the spawning process  
                          were on node 10)
```

If the process that did the *spawn()* had defined its root in its private prefix tree, then this private prefix tree would be inherited by the remote **wc** command, resulting in the same resolution for root (i.e. /) on the local and remote node.

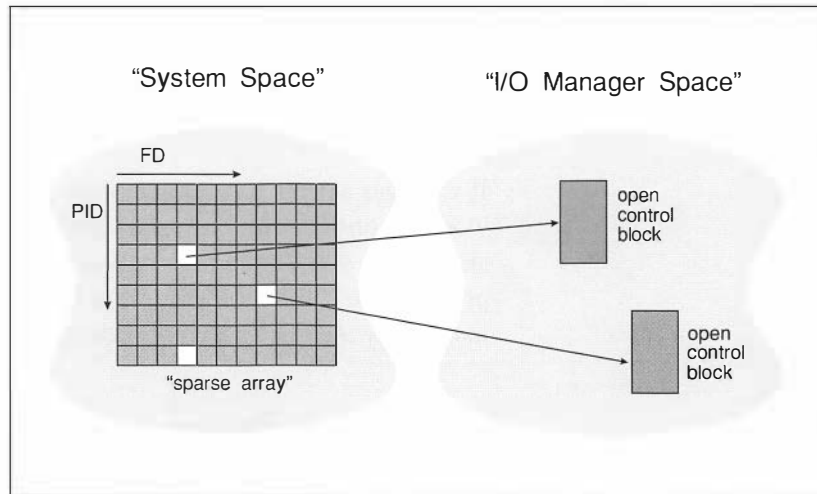
If you pass a name to a remote process you didn't create, then it won't inherit your private prefix tree. To prevent any ambiguity in such cases, you should pass fully qualified network names. QNX provides the C function *qnx_fullpath()*, which maps any pathname into a fully qualified network pathname (i.e. //node/pathname).

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.

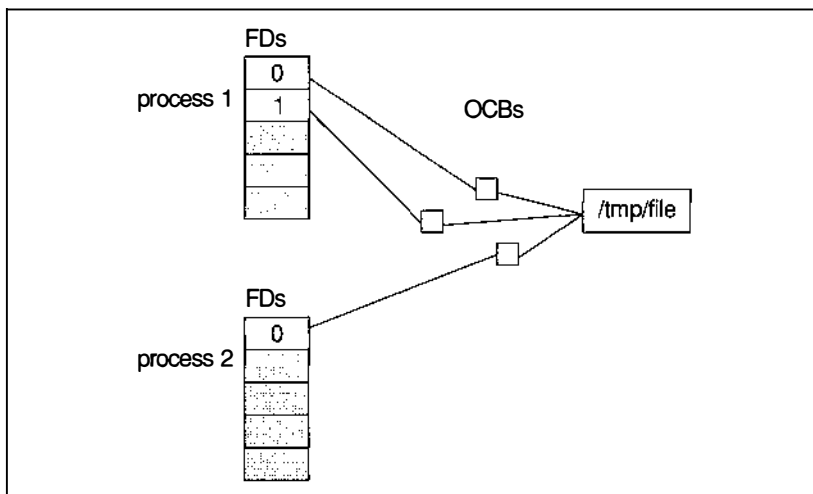


The PID and FD map to an OCB of an I/O Manager.

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.

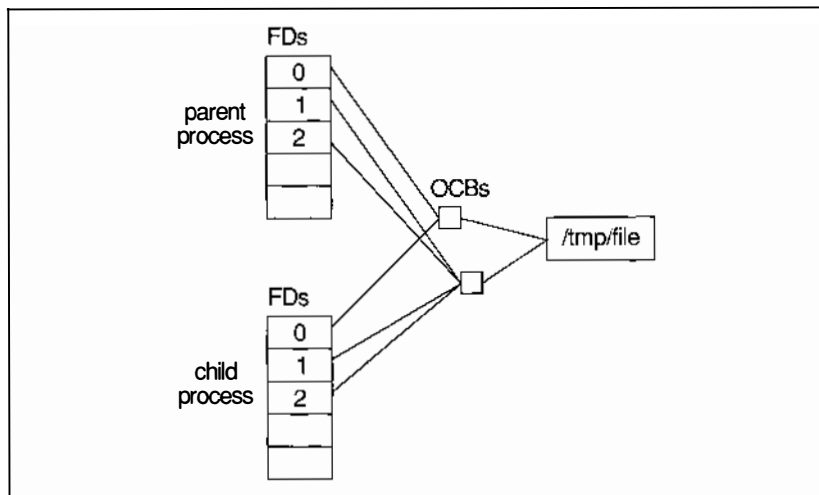
Several file descriptors in one or more processes can refer to the same OCB. This is accomplished by two means:

- 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 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 `fork()`, `spawn()`, or `exec()` by calling the `fcntl()` function and setting the `FD_CLOEXEC` flag.

Chapter 5

The Filesystem Manager

In this Chapter...

Introduction

What is a file?

Regular files and directories

Pipes and FIFOs

Filesystem Manager performance

Filesystem robustness

Raw volumes

Key components of a QNX partition

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 to files.

What is a file?

In QNX, a *file* is an object that can be written to, or read from, or both. QNX implements five types of files; four 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
- **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 fifth filetype, the *character special file*, is managed by the Device Manager.

Date and time stamps

QNX 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

Regular files and directories

Regular files

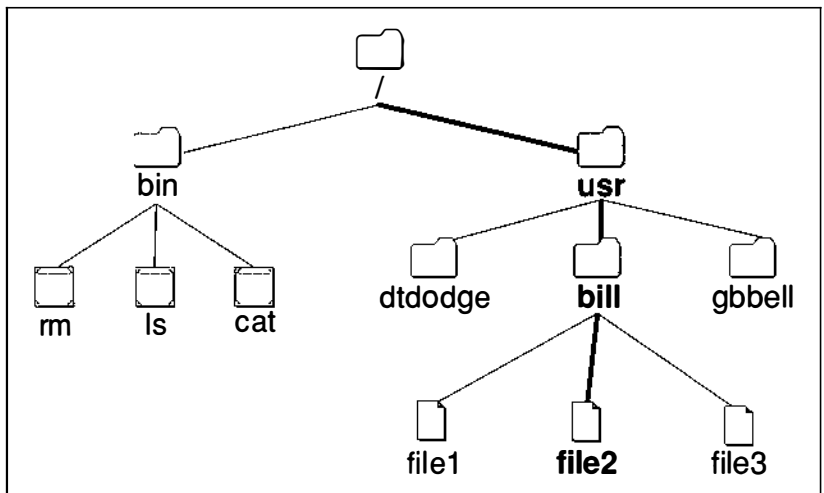
QNX views a *regular file* as a randomly accessible sequence of bytes that has no other predefined internal structure. Application programs are responsible for understanding the structure and content of any specific regular file.

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 section on “Links and inodes”).

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



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

Directory operations Although a directory behaves much like a standard file, QNX 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:

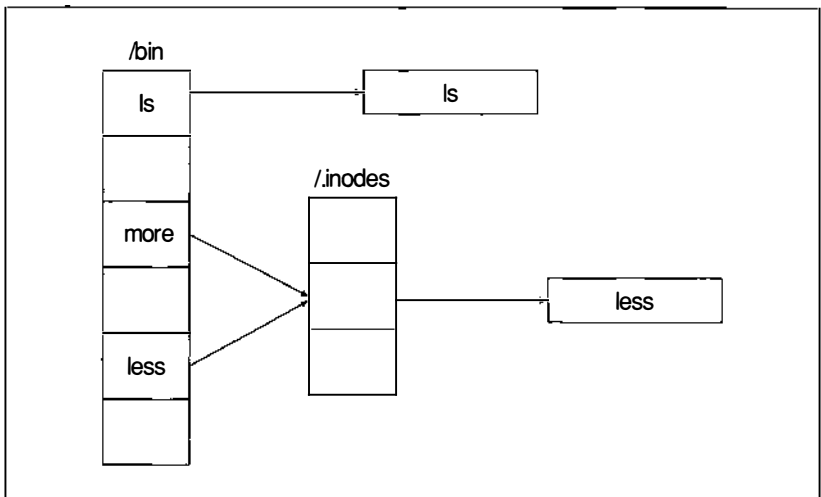
opendir()
readdir()
rewinddir()
closedir()

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.

Links and inodes In QNX, files can have more than one name. Each additional filename is called a *link*. So that links can be supported 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 filename, the inode information (i.e. the non-filename information) is stored in the directory entry for the file. But if the file has more than one link (i.e. more than one filename), 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).

If you want to:	you use the:
create links from within the shell	ln utility
create links from within programs	<i>link()</i> 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 file isn't physically

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:	you use the:
remove links from within the shell	<code>rm</code> utility
remove links from within programs	<code>unlink()</code> C function

Directory links You cannot create links to directories. However, directories have two hard-coded links. There are named:

- . “dot”
- . . “dot dot”

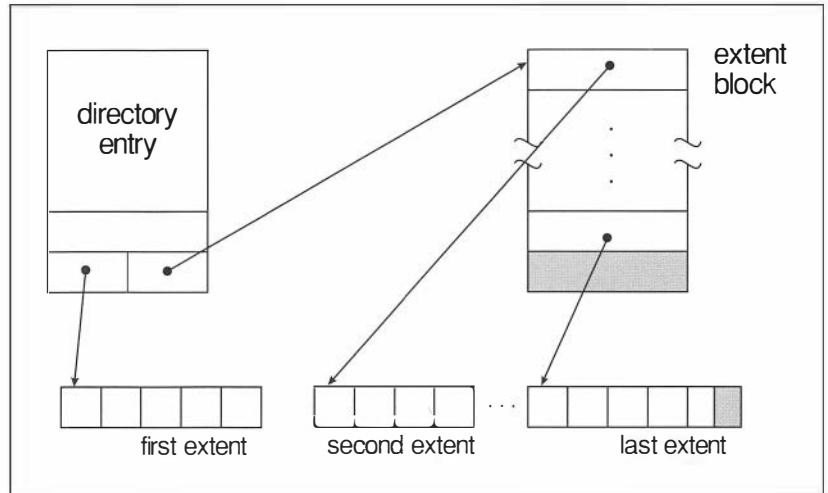
The filename “dot” refers to the current working directory; “dot dot” refers to the parent directory of the current working directory.

Extents

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

Where extents are stored 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.

Separating extents from data allows applications to make highly efficient read and write requests in sizes that are multiples of a sector, without having data cross sector boundaries. This can provide significant benefits to high-speed database applications.



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

Extending files

When QNX 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, QNX 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. QNX chooses the first entry in this table large enough to satisfy the request for a new extent.

Pipes and FIFOs

Pipes

A *pipe* is an unnamed file that serves as an I/O channel between two cooperating processes—one process writes into the pipe, the other reads from the pipe. The Filesystem Manager takes care of buffering the data and synchronizing the two processes (the buffer size is 5120 bytes). A pipe is removed once both its ends have closed.

Pipes are normally used when two processes want to run in parallel while data moves from one process to the other, but there’s no need for bidirectional communication (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 to the standard input of the **more** utility through a pipe.

If you want to:	you 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

FIFOs

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

If you want to:	you use the:
create FIFOs from within the shell	mkfifo utility
create FIFOs from within programs	<i>mkfifo()</i> C function
delete FIFOs from within the shell	rm utility
delete FIFOs from within programs	<i>unlink()</i> C function

Filesystem Manager performance

The Filesystem Manager has a number of features that contribute to high-performance disk access. These features include:

- 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. The buffer cache can be up to 2M in size—the maximum size is configurable at OS build time. If a maximum size isn't set, the buffer cache defaults to $\frac{1}{8}$ of total system memory. (On machines with less than 2M of memory, a smaller fraction is used.)

Read operations are synchronous. Write operations, on the other hand, are usually asynchronous. That is, the Filesystem Manager replies to the client process when the data enters the cache. The data is written to the disk as soon as possible, typically not more 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 filesystem 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 has its priority driven by the priority of the processes that send it messages. When it 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 since the ramdisk is built in, not implemented as a driver. (For information on multi-part messaging, see Chapter 2, The Microkernel.)

Filesystem robustness

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

For example, 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).

In some cases—such as renaming a file to move it to a directory that is full and must grow—all of the above structures must be updated. In these cases, the order of operations has been carefully chosen such that if a catastrophic failure occurs with the operation only partially completed (e.g. during a power failure), the 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.

Filesystem recovery

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. This standard defines a partition table that can define up to four primary partitions on the disk. The table is stored on the first disk block.

Each partition may be given a “type” that is 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
7	OS/2 HPFS
8	QNX 1.x and 2.x (“ qny ”)
9	QNX 1.x and 2.x (“ qnz ”)
77	QNX 4.0
99	UNIX

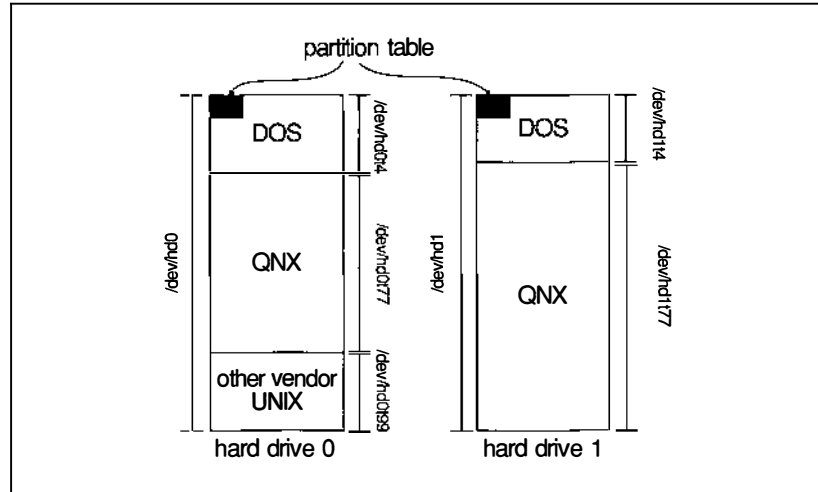
If you want more than one QNX 4.0 partition on a single physical disk, you would use type 77 for your first QNX partition, type 78 for your second QNX partition, and type 79 for your third QNX partition. You can use other types for your second and third partitions, but 78 and 79 are preferred. To mark any of these partitions as bootable, you use the **fdisk** utility.

At boot time, the QNX boot loader (optionally installed by **fdisk**) can 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 hd0	/dev/hd0t4
QNX partition on drive hd0	/dev/hd0t77
UNIX partition on drive hd0	/dev/hd0t99
DOS partition on drive hd1	/dev/hd1t4
QNX partition on drive hd1	/dev/hd1t77

Note that the **t_n** 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**.



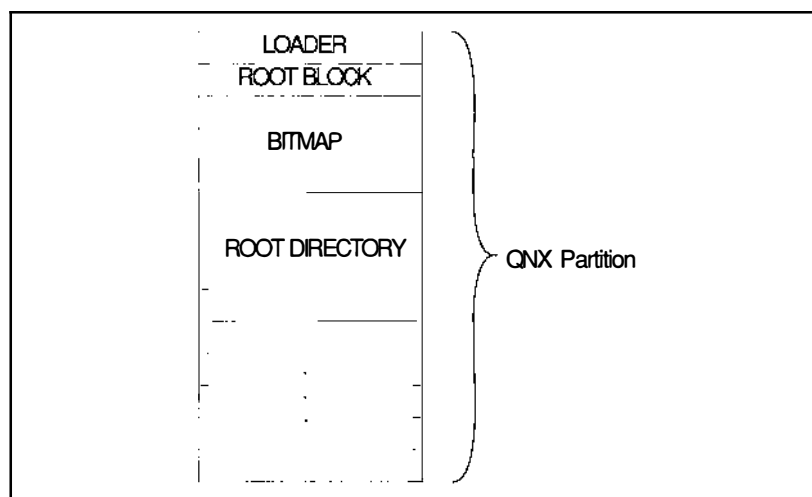
If a disk has been partitioned, you must mount the filesystem only on a block special file that defines a QNX 4.0 partition, not on the block special file that defines the entire raw disk. If you mount the filesystem on the block special file for the entire disk, you'll get a "corrupt filesystem" message when you try to access the disk.

Key components of a QNX partition

There are several key components found at the beginning of every QNX partition that 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

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 is named with a single slash (/). It 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** file and links to the root block inodes of **/.inodes**, **/.boot**, and **/.altboot** files. These entries are provided so programs that report information on filesystem usage will see the entries as normal files.



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

International keyboards

Introduction

The QNX Device Manager (known as **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. In this context, a terminal device represents a bidirectional stream of bytes to a QNX 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. These include:

- timed read operations—*dev_read()*
- asynchronously notifying processes of data available on one or more input devices—*dev_arm()*
- waiting for output to be completely transmitted—*tcdrain()*
- sending breaks across a communications channel—*tcsendbreak()*
- disconnecting a communications channel—*tcdropline()*
- inserting input data under program control—*dev_insert_chars()*
- non-blocking reads and writes—O_NONBLOCK mode of *open()*

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 applications run in *edited* mode. The Shell is a typical example.

Several special control characters may be set in the *termios* control structure to define how **Dev** performs this editing:

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

Line-editing characters vary from terminal to terminal. The QNX console always has 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:

```
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 is based on two members of the *termios* control structure. These are called MIN and TIME. The application can specify a further qualifier of accepting input data when it issues its read request (using *dev_read*); this qualifier is called TIMEOUT.

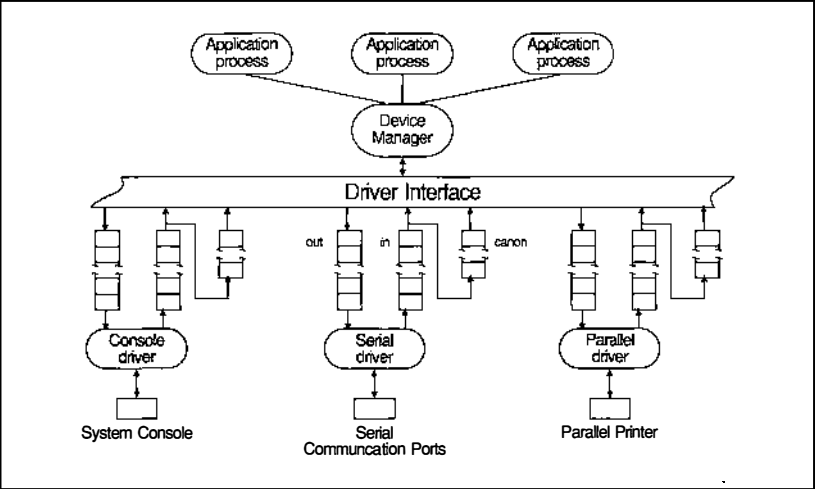
When a QNX process issues a *read* request for *n* bytes of data, these three parameters define when that read request is to be satisfied:

MIN	TIME	TIMEOUT	Description
Ø	Ø	Ø	Return immediately with as many bytes as are currently available (up to <i>n</i> bytes).
M	Ø	Ø	Return with up to <i>n</i> bytes only, when at least M bytes are available.
Ø	T	Ø	Return with up to <i>n</i> bytes when at least one byte is available, or T × .1 second has expired.
M	T	Ø	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 × .1 second.
Ø	Ø	t	RESERVED.

MIN	TIME	TIMEOUT	Description
M	Ø	t	Return with up to <i>n</i> bytes when $t \times .1$ second has expired, or M bytes are available.
Ø	T	t	RESERVED.
M	T	t	Return with up to <i>n</i> bytes when M bytes are available, or $t \times .1$ 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 $T \times .1$ 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 has an affect on 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 a concept called *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.

Switching virtual consoles

The following table gives examples of the <Ctrl>-<Alt>-<n> key chord, which lets you view any available virtual console.

If you want to see:	you enter:
<code>/dev/con1</code>	<Ctrl>-<Alt>-<1>
<code>/dev/con2</code> (if available)	<Ctrl>-<Alt>-<2>
<code>/dev/con8</code> (if available)	<Ctrl>-<Alt>-<8>

Another method of viewing consoles is to use the following key chords, which cycle through only those virtual consoles currently used by an application (i.e. consoles on which there's something to see).

If you want to see:	you press:
the next used console	<Ctrl>-<Alt>-<Enter> <i>or</i> <Ctrl>-<Alt>-<+>
the previous used console	<Ctrl>-<Alt>-<->

<+> is the gray plus key

<-> is the gray minus key

The number of consoles available is specified as a command-line argument to **Dev.con** when it's started. To see how many consoles have been configured in a QNX system, you can use the **ls** utility:

```
ls /dev/con*
```

Changing the screen size

You can also use a key chord to change the size of the displayed characters on the screen, depending on your hardware. EGA and VGA graphics adapters are examples of hardware that permit multiple display-character sizes.

The following table shows you how to change the size of the currently visible console.

If you want to:	you press:
increase the number of displayed rows by selecting a smaller character font	<Ctrl>-<Alt>-< > > ("greater than" key)
decrease the number of displayed rows by selecting a larger character font	<Ctrl>-<Alt>-< < > ("less than" key)

Console-specific functions

In addition to implementing the standard QNX Terminal (as defined in Appendix A), 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:

- direct read of the console screen—*console_read()*
- direct write to the console screen—*console_write()*
- asynchronous notification of significant events—*console_arm()*—such as:
 - display data has changed
 - cursor has moved
 - display size has changed
 - visible console has changed
- control of console size—*console_size()*
- control of visible console—*console_active()*

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 gives the ports names like **/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 results in 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 there is a pending read operation, *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 only one copy of the data is ever made.

These rules—coupled with the extremely small interrupt and scheduling latencies inherent within QNX—result in a very lean input model.

International keyboards

The QNX console driver is capable of responding to a variety of IBM-compatible keyboard layouts. You can use the **kbd** utility to select the appropriate keyboard configuration.

QNX provides a variety of configurations corresponding to the keyboards most commonly used throughout the world. By default, the QNX console driver “assumes” a 101-key USA keyboard layout. The system administrator can use the **kedit** utility to generate custom keyboard layouts.

Some keyboard layouts—such as the French and German layouts—provide for accent keys which, by themselves, don't generate a character. QNX treats these keys as “dead” keys. Pressing a dead key, followed by a second key, modifies the second key, creating an accented character. For example, to create the **ü** character, you press **<">**, followed by **<Shift>-<U>**.

This dead key processing provides typists with a familiar method of composing characters.

Note that you can also generate composed characters by:

- 1) pressing and releasing <Alt>
- 3) typing two characters

For more information, see Appendix A.

Part II

Using the QNX Shell

In this Part...

Using the Keyboard

Command Basics

Working in the QNX Filesystem

Shell Topical Reference



In this part, you'll find a simple introduction to using the QNX command interpreter or Shell. The first three chapters walk you through keyboard usage, command basics, and the QNX filesystem. The fourth chapter, *Shell Topical Reference*, is an alphabetical reference tool containing various entries, from "About the Shell" to "Using Shell commands."

If you get stuck, remember that the index contains entries for all the functions and concepts described in this part.



Chapter 1

Using the Keyboard

In this Chapter...

Entering line-oriented input

Recalling commands

Switching virtual consoles

Using multiple consoles

Changing the screen size

Suspending and resuming output

Rebooting

The keyboard at a glance

Introduction

This chapter describes the standard keyboard conventions in QNX. Remember that some keys may behave differently from how they are described here, depending on how your system administrator has configured your system.

Entering line-oriented input

Line-editing keys

The following table lists the keys you can use for entering line-oriented character input.

	If you want to:	you press this key:
Moving the cursor	move the cursor left one position	← (left arrow key)
	move the cursor right one position	→ (right arrow key)
	move the cursor to the beginning of the line	<Home>
	move the cursor to the end of the line	<End>
Deleting characters	delete the character to the left of the current cursor position*	← (gray arrow key)
	delete the character at the current cursor position	
	cancel the current line (i.e. delete all characters on the line)	<Ctrl>-<X>
Inserting/overwriting characters	toggle between insert mode and typeover mode (default is typeover)**	<Ins>
Submitting a line	submit a line of input	<Enter>

* The gray arrow key may be labeled <Rubout> or <Backspace> on your keyboard. Pressing this key generates a 7F hex (ASCII Rubout), not a 08 hex.

** If you're in insert mode, and you cancel or submit a line, you will be returned to typeover mode.



Your keyboard may not behave as indicated if:

- you're working with an application that has complex requirements for user interaction—the application may take control over how the keyboard works
 - you're working at an attached terminal—the terminal may have keyboard limitations
-

Maximum length of an input line

The maximum length of an input line is normally 256 characters, but this can be decreased or increased by your system administrator. Application programs can impose lower limits.

Entering long input lines

If you enter a single line of input that's too long for your screen to display as a single line, the line may be displayed as several screen lines. The line still behaves as a single input line, however, when you use the standard line-editing keys.

Recalling commands

The QNX Shell lets you recall commands that you've previously entered, then re-execute them. These commands are maintained by the Shell in a circular buffer.

If you want to move:	you press this key:
backward through the buffer	↑ (up arrow key)
forward through the buffer	↓ (down arrow key)

When you recall a command, you're free to alter it using the standard line-editing keys.

Switching virtual consoles

The display adapter and the screen, plus the system keyboard, are collectively referred to as the *console*. To let you interact with several applications all at once, QNX permits multiple sessions to be run concurrently on consoles, using a concept called *virtual consoles*.

Each virtual console can be running a different foreground application that uses the entire screen. The keyboard is attached to the virtual console currently visible. You can switch from one virtual console to another—and thus from one application to another—by entering the following key chords.

If you want to see:	you press:
the next used console	<Ctrl>-<Alt>-<Enter> or <Ctrl>-<Alt>-<+>
the previous used console	<Ctrl>-<Alt>-<->

<+> is the gray plus key
<-> is the gray minus key

Note that you can also “jump” to a specific console using the <Ctrl>-<Alt>-<n> key chord, where *n* is a numeric digit that represents the console number of a virtual console. For more information, see Part I, Chapter 6, The Device Manager.

Using multiple consoles

The system administrator can specify how many virtual consoles are supported on your machine by specifying a command-line argument when the console driver process (**Dev.con**) is started.

The administrator can also specify the program—if any—that is initially launched on each console. By default, the terminal initialization utility (**tinit**) launches a **login** on the first console only, but will be “armed” to launch **logins** on any other console on which you press a key.

This means that console 1 is always available, but that other consoles aren't used unless you specifically request a **login**.

To start a **login** on an unused console, follow these steps:

- 1) Switch to the unused console via <Ctrl>-<Alt>-<n>.
- 2) Press any key. A **login** will be launched.

You can now access the console via any of the cyclical console-switching key chords (e.g. <Ctrl>-<Alt>-<+>) described in "Switching virtual consoles."

When you terminate the session by typing **logout** or **exit**, or by entering <Ctrl>-<D>, the console will once again be idle. It won't appear when you use any of the cyclical console-switching key chords.

Changing the screen size

Depending on your hardware, you can change the number of rows displayed by the currently visible console. Examples of hardware that permit multiple screen character sizes include EGA and VGA graphics adapters.

If you want to:	you press:
increase the number of displayed rows by selecting a smaller character font	<Ctrl>-<Alt>-< > > ("greater than" key)
decrease the number of displayed rows by selecting a larger character font	<Ctrl>-<Alt>-< < > ("less than" key)

Suspending and resuming output

If you want to:	you press:
suspend the display of output	<Ctrl>-<S>
resume the display of output	<Ctrl>-<Q>

Rebooting

To reboot your computer, you use this key chord:

<Ctrl>-<Alt>-<Shift>-



Before entering this command, make sure that no applications or utilities are running on your computer. If there are, files may be left open. Moreover, if you reboot when a critical update is in progress, it's possible that the filesystem would require maintenance (see the section on "Filesystem robustness" in Part I, Chapter 5).

The keyboard at a glance

	If you want to:	you press:
Line editing	move the cursor to the left	←
	move the cursor to the right	→
	move the cursor to the start of a line	<Home>
	move the cursor to the end of a line	<End>
	delete the character left of the cursor	← (gray arrow key)
	delete the character at the cursor	
	cancel an entire line	<Ctrl>-<X>
	toggle between insert/typeover modes	<Ins>
	submit a line	<Enter>
Controlling output	suspend display of output	<Ctrl>-<S>
	resume display of output	<Ctrl>-<Q>
Recalling commands	recall a command	↑ <i>or</i> ↓
Switching consoles	switch to the next virtual console	<Ctrl>-<Alt>-<Enter> <i>or</i> <Ctrl>-<Alt>-<+>
	switch to the previous virtual console	<Ctrl>-<Alt>-<->
	switch to a specific console	<Ctrl>-<Alt>-<n>
Changing the screen size	increase the number of displayed rows	<Ctrl>-<Alt>-< > >
	decrease the number of displayed rows	<Ctrl>-<Alt>-< < >
Rebooting	reboot your computer	<Ctrl>-<Alt>-<Shift>-



Chapter 2

Command Basics

In this Chapter...

Understanding command syntax

Displaying online usage messages

Redirecting input and output

Using wildcards

Entering multiple commands

Understanding command syntax

Syntax structure

Whenever you look up a command in this guide or in the *Utilities Reference*, you'll see a *syntax statement* that summarizes how the command can be used. For most commands, this statement consists of three main parts:

command_name options operands

command_name

The name of the command to be executed. This may be the name of an executable program, such as a utility, or it may be the name of an internal command within the Shell.

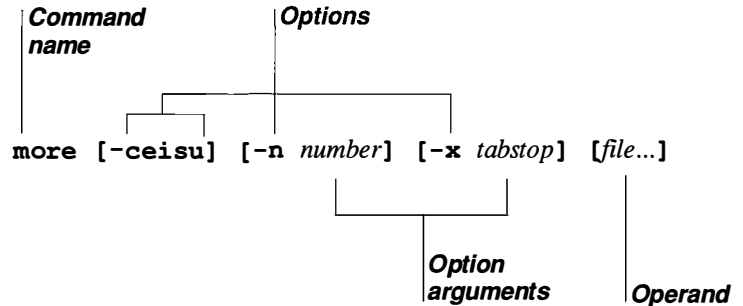
options

These let you specify alternate behavior for a command. Options typically consist of an alphanumeric character preceded by a dash (e.g. “-c”). Some options take an *argument* (e.g. “-n *number*”). If you specify an option that takes an argument, you must include its argument as well.

operands

These let you specify the data the command requires (typically a filename). If a command lets you enter multiple operands, they are usually processed in the order you list them. Unlike options, operands aren't preceded by a hyphen (e.g. “*file...*”).

The following illustration shows a typical syntax statement:



Note that the element:

[-ceisu]

is simply an economical way of indicating that the following options are available:

[-c] [-e] [-i] [-s] [-u]

Key symbols

When reading a syntax statement, you should be aware of these symbols:

... An ellipsis following an option argument or an operand indicates that more than one can be specified. For example, in the above statement for the **more** utility, you see the operand:

file...

This indicates that you can specify more than one file on the command line (e.g. “**more myfile1 myfile2**”).

[] Anything enclosed in brackets ([]) is optional. Options are always enclosed within brackets (e.g. “**[-c]**”). If an *operand* is enclosed in brackets, its use isn’t mandatory (e.g. “**[file...]**” in the above example).

| The OR bar is used between alternatives, indicating that only one can be selected (e.g. “**-a | -f**”).

Grouping options

The following examples of **more** command lines show all the possible ways you can group options and option arguments (each of these command lines would let you display a file called **myfile**):

- 1) You can list each option separately on the command line:

```
more -c -e myfile
```

- 2) You can run several options together using a single dash (-):

```
more -ce myfile
```

- 3) If you run options together with a dash (-), remember that only the last option can take an argument. So you *can* enter this:

```
more -cen10 myfile
```

but you *can't* enter this:

```
more -n10ce myfile
```

- 4) You're free to specify an option argument immediately after the option or separate the two by one or more spaces:

```
more -ce -n10 -x 4 myfile
```

- 5) This example shows all the ways you can group options and their arguments:

```
more -c -e -sn10 -x 4 myfile
```

Arguments containing spaces

An option argument can't contain embedded spaces unless it's enclosed in quotes (" "). So you *can* enter this:

```
-p "match this pattern"
```

but you *can't* enter this:

```
-p match this pattern
```

If an argument contains no spaces (e.g. **matchthispattern**), no quotes are required.

Displaying online usage messages

You can ask for an online usage message for most QNX commands by typing the **use** command. For example, to display the usage message for **more**, you would type:

```
use more
```

If you request usage for a command and the command either doesn't have an executable in the current path or doesn't contain usage message records, an error message is displayed.

Redirecting input and output

Most commands read their input from the standard input file (**stdin**), which is normally assigned to your keyboard, and write their output to the standard output file (**stdout**), which is normally assigned to your display screen. There are occasions, however, when you want commands to:

- read their input from a file or a device other than the keyboard
- redirect **stdout** to a file or to another command

If you want a process to:	you use this symbol:
read from a file, or another device	< (<i>input redirection symbol</i>)
redirect stdout to a file	> (<i>output redirection symbol</i>)
redirect stdout to a file without overwriting the file's contents	>> (<i>output append symbol</i>)
<i>pipe</i> stdout directly to another command	(<i>pipe symbol</i>)

For example, let's look at the **ls** command, which lists the files in a directory. If you want the list of files that **ls** outputs to be redirected to a file called **filelist**, you would enter:

```
ls >filelist
```

Using wildcards

Instead of using a command to work on just one file at a time, you can use *wildcards* (also known as *pattern-matching characters*) to perform operations on several files all at once. You can use the following characters:

- * matches zero or more characters
- ? matches any single character
- [] matches any character (or range of characters separated by a dash) specified within the brackets
- ! excludes characters specified within brackets



Note that directories, as well as files, are matched by these wildcards.

Examples

The following examples show you how you can use wildcards with the **cp** utility to copy groups of files to a directory named **/tmp** (for more information on using **cp**, see the *Utilities Reference*).

If you enter:	cp does this:
cp f* /tmp	copies all files starting with f (e.g. frd.c , flnt)
cp *d /tmp	copies all files ending with d (e.g. frd , fld)
cp fred? /tmp	copies all files beginning with fred and ending with one other character (e.g. freda , fred3)
cp fred[123] /tmp	copies all files beginning with fred and ending with 1 , 2 , or 3 (i.e. fred1 , fred2 , fred3)
cp fred[1-3] /tmp	same as above
cp *. [ch] /tmp	copies all files ending with .c or .h (e.g. frd.c , barn.h)
cp *. [!o] /tmp	copies all files that <i>don't</i> end with .o

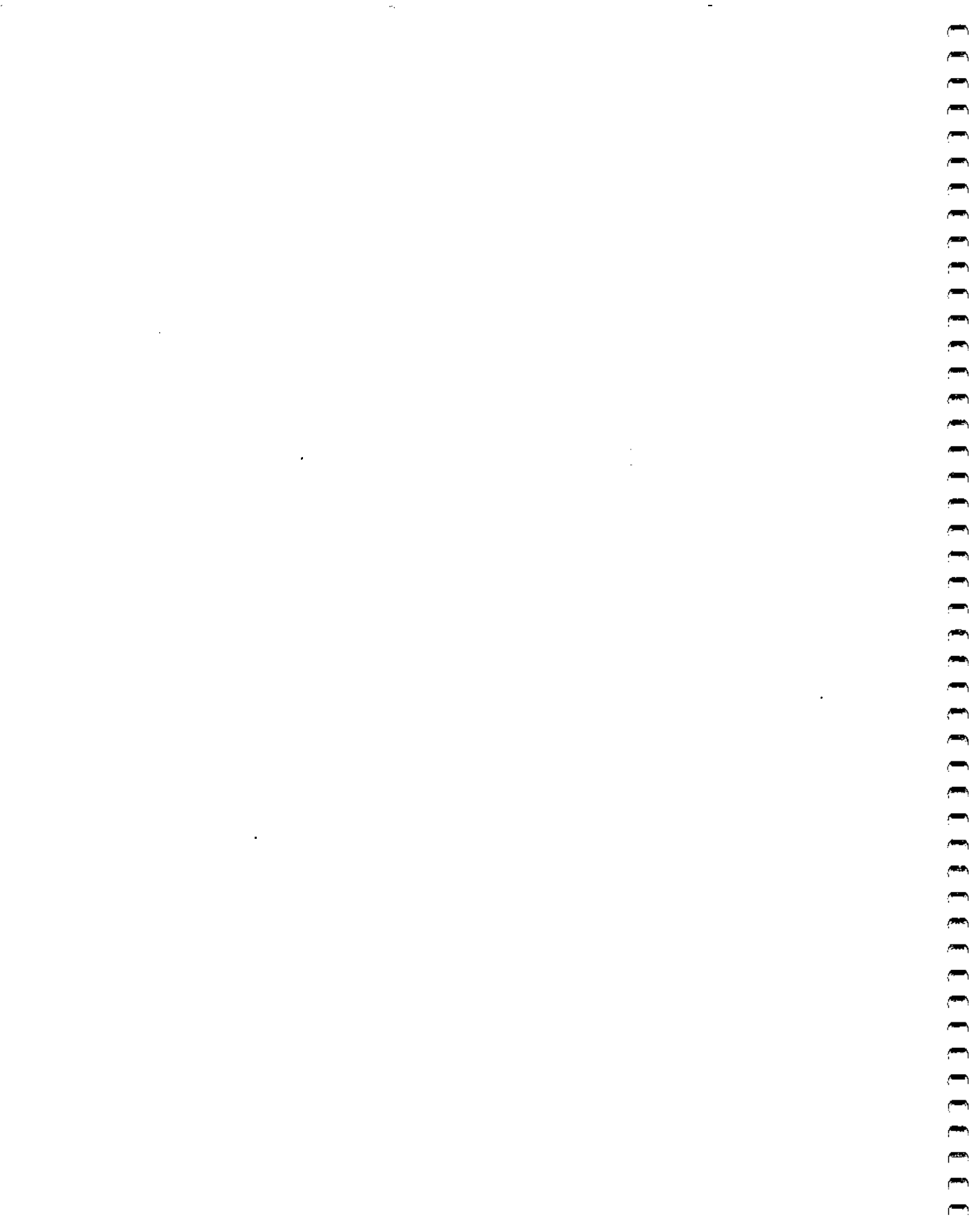
Entering multiple commands

You can enter more than one command at a time by separating your commands with a semicolon (;). For example, if you want to determine your currently working directory and see what it contains, you could enter:

```
pwd;ls
```

Or, let's say you want to make a copy of a file in your current working directory, then see the directory's updated contents. You could enter something like:

```
cp myfile myfile2;ls
```



Chapter 3

Working in the QNX Filesystem

In this Chapter...

Introduction

Filenames

Pathnames

A typical QNX filesystem

Commonly used commands

Introduction

QNX comes with a rich set of utilities and shell commands that let you perform a wide range of operations on files and directories. To help you work with these commands, this chapter provides essential information on filename and pathname conventions in QNX; it also describes the directory structure of a typical QNX filesystem.



For more information on the QNX filesystem, see Part I, Chapter 5, The Filesystem Manager.

Filenames

A *filename* is the symbolic name that lets you identify and access a file. In QNX, a standard filename may be up to 48 characters in length.

You can create filenames with all the characters in the computer's character set except for the slash (/) and the ASCII NUL character.

Portable filenames

If you anticipate moving your programs and data to other systems, you can make your files more portable by naming them only with those characters defined by POSIX for filenames:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789_.-
```



You can't use the dash (-) as the first character of a portable filename.

Filename suffixes

Filename suffixes (e.g. “.txt” in **myfile.txt**) have no special meaning. The “.” character is treated the same as any other character in a filename. Applications often use filename suffix conventions, however. For example, the **make** utility understands .c, .o, .y, etc.

Pathnames

QNX lets you group related files into directories. Directories can also contain other directories, so that you can implement a hierarchical filesystem.

Each filesystem has its own *root directory*. The root is at the top of the directory hierarchy, and it is where QNX starts looking for other directories and files. The typical name for the root directory of the default filesystem on a hard disk is slash (/).

If a hard disk has more than one filesystem, the choice of names for filesystems is up to the system administrator. These may include names such as **/hd2**, **/hd3**, etc., or perhaps **/usr2**, **/usr3**, etc.

To locate a file, you specify the *path* from the root directory to the file. Other directories in the path to a file are also separated by slashes. Thus, to get to the file **proc.h** in the **/usr/lib/** directory, you would specify this path:

```
/usr/lib/proc.h
```

Network pathnames

You sometimes need to specify which computer (or “node”) on the network contains the file you want. To specify a node, you use a leading double slash (//), followed by the node number of the node. So, if you wanted to access **/usr/sam/file** on node 24 of a network, you would enter:

```
//24/usr/sam/file
```

If you don’t explicitly specify a node, QNX uses the currently defined root.

Current working directory

To save you from constantly specifying complete pathnames to access a file, QNX provides a shortcut—the *current working directory*.

The advantage of having a current working directory is that you don’t have to specify a fully qualified path to access files in or below it. For example, if you’re positioned in the directory **/usr/gbbell** and you want to access the file **prog1.c** you *wouldn’t* need to specify:

```
/usr/gbbell/procl.c
```

Instead, you could specify only the filename itself:

```
procl.c
```

Home directory

When you first log in to QNX, your current working directory is set to a directory that has been assigned to you by your system administrator. This directory is known as your *home directory* and is your initial current working directory.

No matter where you are, you can always return to your home directory by entering the following command:

```
cd
```

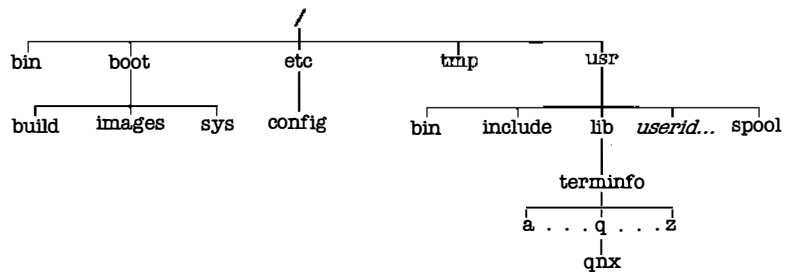


For more information on the pathname space in QNX, see Part I, Chapter 4, I/O Namespace.

A typical QNX filesystem

QNX 4.0 follows most of the UNIX conventions for storing standard files. For example, system executables are stored under **/bin** and temporary files are found under **/tmp**.

The following diagram gives an overview of the general directory structure in a typical QNX installation.



The following table summarizes the types of files you'll find under the main directories:

Directory:	Description:
/bin	system executables
/boot	OS image Makefile
/boot/build	build files to make images (the make utility reads these)
/boot/images	OS image files
/boot/sys	system processes needed at boot time
/etc	initialization and other files
/etc/config	sysinit and configuration files
/tmp	default location for temporary files
/usr/bin	non-system executables
/usr/include	header (.h) files for the C compiler
/usr/lib	libraries for the C compiler
/usr/lib/terminfo/q/qnx	console files (terminfo)
/usr/spool	configuration and work files for system spoolers
/usr/userid	user directories

Commonly used commands

The following table lists the utilities you use to perform common operations. For more information, see the *Utilities Reference*.

	If you want to:	you use this utility:
Working with directories	change your current working directory	cd
	create directories	mkdir
	remove directories	rmdir
	print your current working directory	pwd
	list the contents of a directory	ls
Working with files	compare two files	diff
	concatenate files	cat
	copy files	cp
	count the number of words, lines, or bytes in a file	wc
	delete files	rm
	display the contents of a file	more, less, cat
	move files	mv
	print the contents of a file	lp
	rename files	mv
	search for string patterns in files	grep
	sort or merge lines in a file	sort
Other commands	display who is on the system	who
	find files	find

Chapter 4

Shell Topical Reference

In this Chapter...

About the Shell

Command lists

Command substitution

Environment variables

Expansion

Invocation modes

Pipes

Quoting

Redirection

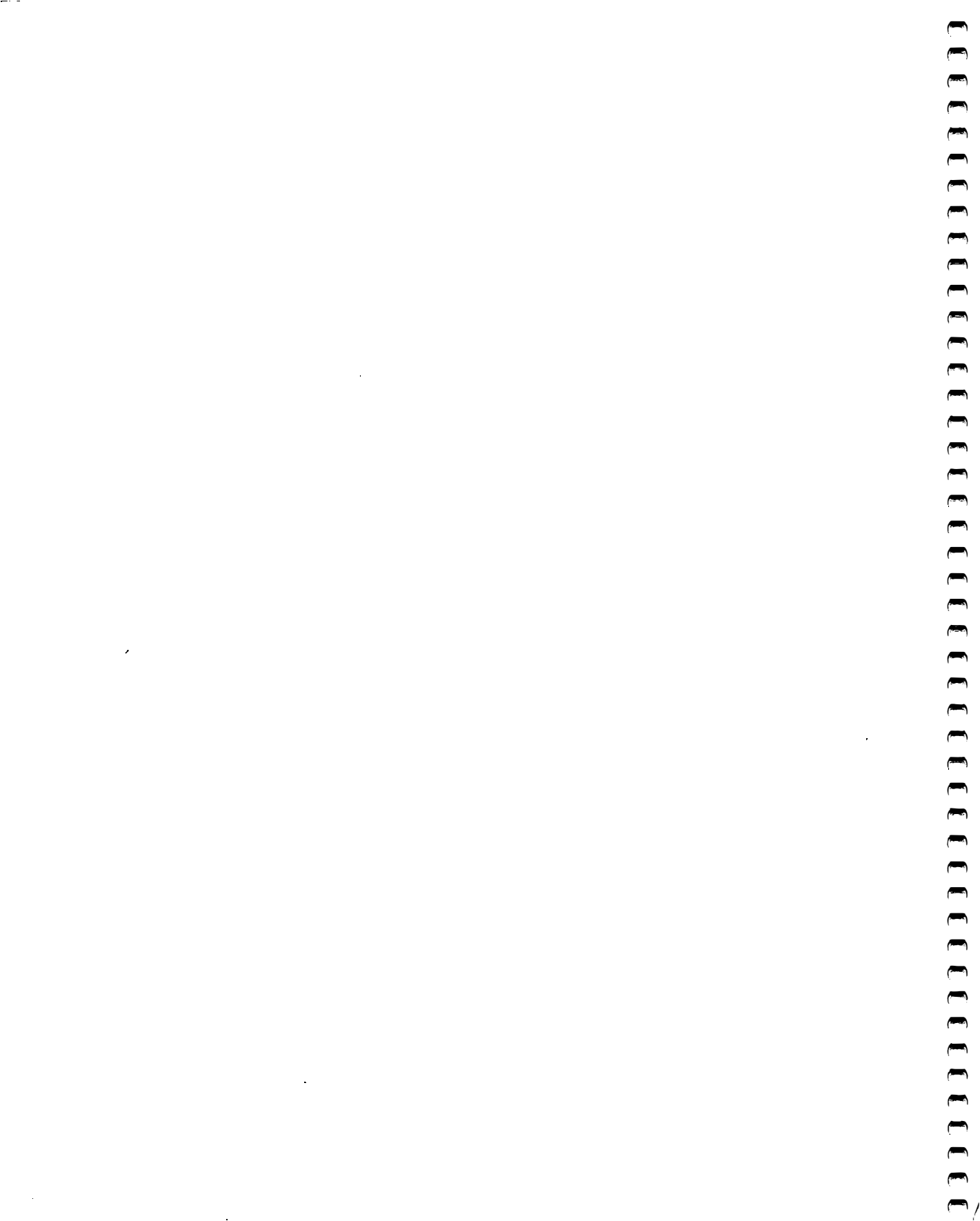
Remote execution

Reserved words

Signals

Terminology

Using Shell commands



About the Shell

The QNX Shell is a *command interpreter* that acts as the interface between the user and the QNX operating system.

The QNX Shell is based on the shell specified in the IEEE POSIX standard. The POSIX shell definition is derived from the AT&T System V Shell, which in turn is based on the Version 7 Bourne Shell (the Korn shell is also based on the Version 7 Bourne Shell). As a result, the QNX Shell shares a base set of functionality with these historical shells—if you have ever worked with any of them, you should soon feel at home with the QNX Shell.

The QNX Shell has many capabilities, including:

- local and remote command execution
- filename expansion
- input and output redirection
- command substitution
- parameter expansion
- a full shell programming language

These capabilities are discussed throughout the various entries in this topical reference.

Command lists

A *command list* is a sequence of commands or pipelines, or both. You can separate the components in a command list by either of these control operators:

- `;` (used for sequential lists)
- `&` (used for background execution)

or by either of these logical control operators:

- `&&` (used for AND lists)
- `||` (used for OR lists)

Sequential lists

If you separate the commands in a list with the `;` control operator, the commands are executed sequentially, one after the other. For example:

```
$ cd /tmp; ls; ls |wc -l
doit.awk      junk      temp      xfile
4
```

This is identical to entering the commands on three separate lines except that the Shell outputs only a single prompt once all the commands have finished executing.

Background lists

If you terminate a single command with the `&`, the command is executed in the background (asynchronously). You can also use this technique to execute a grouped command list in the background:

```
(cd /tmp; ls; ls |wc -l) &
```

For more information, see “Grouping command lists” below.

AND/OR lists

The `$$` and `||` operators allow a list to execute depending on the exit code of the preceding list:

If you use:	the next list is executed only if:
<code>&&</code>	the preceding list succeeded (zero exit code)
<code> </code>	the preceding list failed (nonzero exit code)

Grouping command lists

You can group command lists as follows:

`( list )` Executes *list* in a subshell.

`{ list ; }` Executes *list* in the current process environment.

Command substitution

Command substitution allows the standard output of a command to be substituted in place of the command itself. You can perform command substitution in either of two ways:

- by enclosing the command in parentheses and preceding it with a dollar sign:

`$(command)`

- by enclosing the command in backquote characters:

``command``

For example, `echo $(date)` could produce:

Thu Apr 11 10:23:12 EDT 1991

Note that any trailing sequence of one or more `<newline>` characters is deleted before the output is substituted.

Environment variables

Environment variables are named parameters used by the Shell. You can change these variables to modify the shell environment. When invoked, the Shell sets default values for the **IFS**, **PATH**, **PS1**, and **PS2** variables.

CDPATH The directory search path used by the **cd** utility.

HOME The pathname of the user's home directory

IFS (Input Field Separators) A string treated as a list of characters that are used for splitting fields. The default for **IFS** is:

<space><tab><newline>

LOGNAME The user's **login** name.

PATH The directory search path used by the Shell for locating executable programs and shell scripts. For non-superusers, the default for **PATH** is:

:/bin:/usr/bin

PS1 The primary prompt string (default is "**\$** ").

PS2 The secondary prompt string (default is "> ").

SHELL The pathname of the user's preferred shell.

TERM The terminal type.

TMPDIR The pathname of a directory that can be used by utilities that need a place to create temporary files.

TZ The timezone setting.

See also: The section on "Named parameters" in the "Expansion" entry.

Expansion

Arithmetic expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting the resulting value. The syntax for arithmetic expansion is:

$\$(expression)$

For example, the following would result in the integer division of 365 by 12:

$\$(365/12)$

The types of arithmetic operations that can be processed are similar to those available in the C programming language.

Filename expansion

You use most QNX commands for manipulating files in one way or another. As such, the Shell has a filename “shorthand” that can be used to specify the files that a particular command is to operate on. This “shorthand” consists of these characters:

*** ? []**

If one or more of these shorthand characters is used in a command line, each word containing the characters is treated as a “pattern” and is replaced with the list of filenames matching the pattern. Filenames that begin with a dot (.) aren’t included in the generated list unless a dot is explicitly included as the first character of the pattern.

***** Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the enclosed characters. If you specify two characters separated by a dash (–), this matches the characters and any characters that fall between them (e.g. **[a–c]** matches **a**, **b**, and **c**).

[!...] Matches any character except for the enclosed characters.



Note that directories, as well as files, are matched by these shorthand characters.

Examples

The following example copies all files beginning with **fred** and ending with any digit from 1 to 3 (i.e. **fred1**, **fred2**, **fred3**) to the **/tmp** directory:

```
cp fred[1-3] /tmp
```

This next example copies all files ending with **.c** or **.h** (e.g. **frd.c**, **barn.h**, etc.):

```
cp *. [!ch] /tmp
```

This next example copies all files that *don’t* end with **.o**:

```
cp *. [!o] /tmp
```

Parameter expansion

The QNX Shell supports three kinds of parameters:

- positional parameters
- special parameters
- named parameters (or environment variables)

Parameter expansion is performed on words beginning with **\$** and enclosed in braces “**{ }**”. The word needn’t be enclosed in braces if it’s used in an unambiguous context.

Positional parameters

Positional parameters are the parameters specified on the command line of the Shell and are accessed using the corresponding ordinal number. That is, **\$1** expands to the first argument, **\$2** expands to the second, and so on. Positional parameters can be assigned values using the special built-in command **set**.

Special parameters

The following special parameters are automatically set by the Shell and provide some commonly desired functionality for shell programming. The special parameters and the values to which they expand are as follows:

- \$*** All the positional parameters starting from one. If the expansion occurs within double quotes, a single word with the value of each parameter separated by a **<space>** is generated. If the named parameter **IFS** is set, the words are separated by the first character of its value. (For more information, see the entry on “Environment variables.”)
- \$@** All the positional parameters starting from one. If the expansion occurs within double quotes, the value of each parameter expands to a separate word.
- \$#** The number of positional parameters in decimal.
- \$?** The exit status of the last foreground command (i.e. the exit status of the last synchronous process or pipeline).
- \$\$** The process ID, in decimal, of the invoked shell.
- #!** The process ID of the last background command.
- \$0** The name of the shell script, or the Shell itself.

Named parameters

Named parameters are values represented by a *name* as defined below. The named parameter is simply expanded into its corresponding value.

Note that if a named parameter is exported (via the **export** built-in), it becomes an environment variable.

The simplest form of named parameter expansion is:

`$name` or `${name}`

where the value, if any, of *name* is substituted. A parameter expansion can be modified by one of the following methods:

`${parameter:-word}`

If *parameter* is unset or null, *word* is substituted. Otherwise, the value of *parameter* is used.

`${parameter:=word}`

If *parameter* is unset or null, *word* is assigned to *parameter*. Otherwise, the value of *parameter* is used as is. (Note that positional parameters and special parameters cannot be assigned in this way.)

`${parameter:? [word]}`

If *parameter* is unset or null, *word*—or an error message from the Shell if *word* is omitted—is output to standard error. Otherwise, the value of *parameter* is used.

`${parameter:+word}`

If *parameter* is unset or null, the null string is substituted; otherwise *word* is used.



In the parameter expansions shown above, using the colon results in a test for the parameter being unset or null. If you omit the colon, the test is only for the parameter being unset.

`${#parameter}`

The number of characters in the value of *parameter* is substituted.

`${parameter%word}`

Strip the smallest suffix matching *word* from *parameter* before substituting *parameter*.

`${parameter%%word}`

Strip the largest suffix matching *word* from *parameter* before substituting *parameter*.

`${parameter#word}`

Strip the smallest prefix matching *word* from *parameter* before substituting *parameter*.

`${parameter##word}`

Strip the largest prefix matching *word* from *parameter* before substituting *parameter*.

Tilde expansion

A tilde prefix consists of a tilde character (~) at the beginning of a word, followed by all the characters preceding the first slash (/) in the word, or all the characters if there is no slash. The tilde prefix is treated as a possible login name and is expanded into the pathname of the home directory associated with that login name. If the login name is null (i.e. the tilde prefix contains only the tilde character), the tilde prefix is expanded into the value of the **HOME** environment variable.

For example, if your home directory is `/usr/fred`, the following lines are equivalent:

```
cat /usr/fred/myfile
cat ~/myfile
```

Invocation modes

The following invocation modes are available:

```
sh [-aCefnuvx] [command_file [argument...]]
```

```
sh -c [-aCefnuvx] command_string  
      [command_name [argument...]]
```

```
sh -i [-aCefuvx] [command_file [argument...]]
```

```
sh -s [-aCefnuvx] [argument...]
```

Examples: Invoke a subshell from the command line:

```
sh
```

Invoke a subshell with a command file:

```
sh /etc/backup
```

Invoke a subshell with a command string:

```
sh -c "cd /etc; ls -alF"
```

By default, the Shell is invoked interactively with its standard input and standard error output attached to the invoking terminal or console. To change how the Shell executes shell commands and where it reads them from, you use these options:

- c** Read commands from *command_string*. The special parameter **\$0** is set to *command_name* and the positional parameters (**\$1**, **\$2... \$n**) are set to the remaining *arguments*.
- i** Be “interactive.” The Shell is also invoked interactively if no operands are specified and the Shell’s standard input and standard error are attached to a console or terminal.
- s** Read commands from the standard input. If no operands are given and the **-c** option isn’t specified, the Shell is invoked as if the **-s** option had been specified.

For information on the options **-C**, **-a**, **-e**, **-f**, **-n**, **-u**, **-v**, and **-x**, see the description of the **set** utility in the section, “Special built-in commands.”

Pipes

A *pipe* is a mechanism built into the QNX filesystem that lets you connect the output of one program to the input of another without having to resort to a clumsy temporary file. A *pipeline* is a connection of two or more programs through pipes. The special character `|` informs the shell that a pipe is to be set up.

In the following example, the output of `ls` is piped to `wc` in order to count the number of lines in a file listing:

```
$ ls |wc -l
```

You can have as many programs in a pipeline as you like, as the following examples illustrate.

Count the number of times a particular user is logged on to this node:

```
$ who |grep steve |wc -l
```

Count the number of times that same user is logged on anywhere on the network. Also, save the output of the `who -a` command in the file `/tmp/whonet`:

```
$ who -a |tee /tmp/whonet |grep steve |wc -l
```

Count the number of nodes on the network and save the output of `sin -h net` in the file `/tmp/netinfo`:

```
$ sin -h net |tee /tmp/netinfo |sed -e '/-----/, $d' |wc -l
```

Quoting

You use quoting to suppress the special meaning of certain characters and words to the Shell. You can use quoting to:

- preserve the literal meaning of certain characters
- prevent reserved words from being recognized as such
- prevent parameter expansion and command substitution within “here-document” processing

To quote a single character, sequences of characters, or words, you enclose them in single or double quotes. If only a single character needs to be quoted, it can be preceded with the escape character (`\`).

You *must* quote the following characters if they are to represent themselves:

```
| & ; ( ) $ ` \ " ' <space> <tab>
<newline>
```

You *might* need to quote the following characters, depending on their context within a shell command:

```
* ? [ # ~ = %
```

Escape character (backslash)

A backslash character (`\`) that isn't quoted preserves the literal value of the next character, unless the next character is a `<newline>`. If a `<newline>` follows the backslash, the Shell interprets this as *line continuation*.

Single quotes

Enclosing characters and words in single quotes (`' '`) preserves the literal meaning of each character and reserved word within the single quotes. A single quote cannot be included with single quotes. For example:

```
echo 'What\'\'s up Doc?'
```

results in:

```
What's up Doc?
```

Double quotes

Enclosing characters and words in double quotes (" ") preserves the literal meaning of all characters and words within the double quotes, with the exception of the \$, ', and \ characters:

This character:	retains its special meaning:
\$	of introducing parameter expansion, command substitution, and arithmetic expansion
'	of introducing the alternate form of command substitution (see "Command substitution")
\	as the quote character

To include a double quote character within double quotes, you precede it with the escape character. For example:

```
toon=Bugs
echo "$toon: \"What's up Doc?\""
```

results in:

```
Bugs: "What's up Doc?"
```

Shell prompts

The Shell always prompts for input. By default, it displays this prompt:

```
$
```

This prompt is displayed whenever the Shell is waiting for input at the command line. The Shell also has a secondary prompt, which by default is set to:

```
>
```

The Shell displays this secondary prompt whenever a <newline> is entered and the Shell requires further input to complete the command (unless the <newline> has been escaped).

In the following example, the Shell presented the secondary prompt because the string wasn't terminated with a closing quotation mark.

```
$ echo "QNX 4.0, it's POSIX
> and a whole lot more."
```

Redirection

When a command is run, part of its environment consists of standard files set up for its use. These files, standard input, standard output, and standard error output (**stdin**, **stdout**, **stderr**), are usually attached to the active terminal. When the Shell executes a command, the command inherits a copy of the Shell's file descriptors without having to open them again. Before a command is executed, its input and output can be redirected using a special notation interpreted by the Shell. The general syntax used for redirection is:

`[n] redir-op file`

where *n* is an optional file descriptor number, *redir-op* is one of the redirection operators described below, and *file* is usually the name of a file.

Redirecting input	<code>[<i>n</i>] <<i>file</i></code>	Opens <i>file</i> for reading on the specified file descriptor (<i>n</i>) or on standard input if the file descriptor isn't given.
Redirecting output	<code>[<i>n</i>] ><i>file</i></code> <code>[<i>n</i>] > <i>file</i></code>	Opens <i>file</i> for writing on the specified file descriptor (<i>n</i>) or on standard output if the file descriptor isn't given. If the file exists, it's overwritten. If the file does not exist, it's created. If the <i>noclobber</i> option is set (see the set command) and the file already exists, <code>[<i>n</i>] ><i>file</i></code> will fail, whereas <code>[<i>n</i>] > <i>file</i></code> won't.
Appending output	<code>[<i>n</i>] >><i>file</i></code>	Opens <i>file</i> for writing on the specified file descriptor (<i>n</i>) or on standard output if the file descriptor isn't given. If the file exists, output is appended to the end of the file. If the file does not exist, it's created.
The here-document	<code>[<i>n</i>] <<[-] <i>word</i></code>	Reads the specified file descriptor <i>n</i> , or the standard input if a file descriptor isn't given, until a line that is the same as <i>word</i> is found or the end-of-file is reached.

The resulting document becomes the standard input for the next command to be executed, hence the term “here-document.”

If no characters in *word* are quoted, all lines of the here-document have parameter expansion, arithmetic expansion, and command substitution performed on them. In this case, you must use the backslash character (\) to quote these characters:

\ \$ `

If the form `[n]<<-word` is used, leading tabs are stripped from the input lines and from the line containing the delimiter *word*.

The following example will output the text between the first line, which contains the here-document operator (<<), and the last line, which consists of the delimiter (**THE_END**). The text between these two lines will be output without the leading tabs. This example could be used instead of a sequence of `echo` commands.

```
cat <<-THE_END
    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.
THE_END
```

Duplicating an input or output file descriptor	<code>[n]<&digit</code> <code>[n]>&digit</code>	The specified file descriptor <i>n</i> , or the standard input if <i>n</i> isn't given, is made to be a copy of the input file descriptor specified by <i>digit</i> . The second form <code>[n]>&digit</code> descriptor behaves similarly for duplicating an output file descriptor.
Opening a file descriptor for reading and writing	<code>[n]<>file</code>	Opens <i>file</i> for both reading and writing on the specified file descriptor or on standard input if the file descriptor isn't given.

Closing a file descriptor

[n]<&- Closes the specified file descriptor *n*. If a file descriptor isn't given, the standard input is closed. The second form **[n]>&-** behaves similarly for closing an output file descriptor.

Remote execution

The Shell supports QNX's integrated networking by providing a syntax for executing programs on remote nodes. This syntax can be in one of two forms:

onnode *node command [argument...]*

OR

//node *command [argument...]*

In the following example, the **sin** command reports the system information of the node the command is run on:

```
$ sin info
Node  CPU  Machine Speed  Memory  Ticksize  Monitor  Flags
  1  486/487      AT  2109 12006k/15871k 10ms  VGA Color -----8P

Alloc  Handlers Names Sessions Procs Timers Nodes Boot
10120      32   30      32   150    40 43690  Net

Locators:
1
```

In this next example, the **sin** command is preceded by the **onnode** command in order to report the system information of node number 2:

```
$ onnode 2 sin info
Node  CPU  Machine Speed  Memory  Ticksize  Monitor  Flags
  2  386/0      AT   583 4977k/ 5760k 10ms  VGA Color -----E-P

Alloc  Handlers Names Sessions Procs Timers Nodes Boot
13892      32   30      32   150    40  100  Net

Locators:
1
```

The second syntax form is a bit of a short cut. The following command:

```
$ //2 sin info
```

is identical in functionality to the above example using **onnode**.

Reserved words

Reserved words are words that have special meaning to the Shell. The following words are reserved:

!	elif	fi	in	while
case	else	for	then	{
do	esac	if	until	}
done				

Shell scripts

A *shell script* is a text file that contains a sequence of shell commands. A shell script may be executed in either of two ways:

- from within an interactive shell as a command
- in batch mode as a command-line argument to the Shell (for more information, see the entry on “Invocation”).

The example on the following page is an implementation of the **which** utility as a QNX shell script. Of special interest within this shell script is the first line:

```
"#! /bin/sh"
```

This is a special syntax that informs the invoking shell to spawn another shell (in this case, **/bin/sh**) to execute the shell script. For example, let's say the user's current shell is the Korn shell (**ksh**) and this shell script has been invoked by entering **which ls** at the command line. The Korn shell first reads this line, then invokes the QNX shell (**/bin/sh**) to execute the shell script.

The first line of this script informs the invoking shell to spawn another shell (in this case, `/bin/sh`) to execute the shell script.

In a shell script, any text following the special character `#` is treated as a comment and hence ignored. Blank lines are also ignored.

```
#!/bin/sh

#
# which: display which command in PATH is executed
#
# Use: which command
#

oldpath=$PATH
PATH=/bin:/usr/bin

case $# in
0)    echo 'which command' 1>&2; exit 2
esac

for path in `echo $oldpath | sed 's/^:./:./
              s/::/:.:/g
              s/.$:./
              s/:/ /g'`
do
    if test -f $path/$1      # Is the command here?
    then
        echo $path/$1      # Yes, so display it and exit.
        exit 0
    fi
done

exit 1                        # Didn't find it...
```

Signals

When a command is an asynchronous command list (i.e. it is run in the background using the control operator `&`), the Shell prevents the SIGQUIT and SIGINT signals from interrupting the command. In the case of foreground commands, the Shell allows the same signals as its parent. Signals can also be *trapped* using the special built-in command `trap`.

Terminology

AND-OR-list

A sequence of one or more pipelines separated by the operators `&&` and `||`.

blank

Any one of the characters currently set in the **IFS** environment variable. The defaults are usually the `<tab>`, `<space>`, and `<newline>` characters.

command list

A sequence of one or more AND-OR-lists separated by the operators `;` and `&`.

comment

Any word beginning with a `#` character. The word and all characters following it are ignored until a `<newline>` is detected.

control operator

One of the following symbols:

`&` `&&` `( )` `;` `;;` `<newline>` `|` `||`

name

A sequence of letters, digits, or underscores beginning with a letter or an underscore.

parameter

A *name*, a digit, or any of these characters:

`*` `@` `#` `?` `-` `$` `!`

If a named parameter is exported, it becomes an environment variable.

pipeline

A sequence of one or more commands separated by the control operator `|`. The standard output of all but the last command is connected to the standard input of the next command.

redirection operator

One of the following symbols:

```
< > >| << <<- >>
<& >& <> <&- >&-
```

simple-command

A sequence of optional variable assignments and redirections, optionally followed by words and redirections, and terminated with a control operator.

word

Any sequence of characters—other than a control operator or redirection operator—that the Shell treats as a single unit when reading input.

Using Shell commands

A shell command is one of the following:

- control construct
- regular built-in utility
- special built-in command
- executable utility
- shell script

Commands invoked within the Shell have the following precedence:

- 1) special built-in commands
- 2) functions
- 3) regular built-in utilities
- 4) executable utilities and shell scripts

For example, when you enter:

```
umask
```

the Shell will execute its regular built-in **umask** instead of the executable utility **umask** since regular built-ins have higher precedence than executable utilities.

The Shell special built-ins, functions, and regular built-ins are all resolved and executed within the Shell process. Executable programs and shell scripts are found within the QNX filesystem using the **PATH** environment variable and are spawned as subprocesses.

Control Constructs

The **case** conditional construct

```
case word in
    pattern1)          list ;;
    pattern2 | pattern3) list ;;
    ...
esac
```

The **case** command executes the *list* associated with the first *pattern* that matches *word*. Patterns are the same as those used for filename generation except that the rules concerning the slash character and a leading dot don't apply.

The **if** conditional construct

```
if expr
then
    list
[elif expr
then
    list] ...
[else
    list]
fi
```

The **if** *expr* is executed and if its exit status is zero, the **then** *list* is executed and the **if** construct is complete.

If the exit status of **if** *expr* is nonzero, however, each **elif** *expr* is executed in turn. If the exit status of an **elif** *expr* is zero, its corresponding **then** *list* is executed.

If the exit status of an **elif** *expr* is nonzero, the **else** *list*, if any, is executed.

The **for** loop

```
for name [ in word... ]
do
    list
done
```

The variable *name* is set to each *word*, in turn, and *list* is executed each time. If the **in word...** is omitted, then the **for** command executes *list* once for each positional parameter set (i.e. the equivalent of **for name in "\$@"**)

The **while** loop

```
while list1
do
    list2
done
```

The **while** command executes *list1* and if *list1*'s exit status is zero, *list2* is executed and the loop is repeated. Otherwise, the loop is exited.

The **until** loop

```
until list1
do
    list2
done
```

The **until** command executes *list1* and if *list1*'s exit status is nonzero, *list2* is executed and the loop is repeated. Otherwise, the loop is exited.

The function definition
command

```
name ()
{
    list
}
```

Define a function *name* that is used by the Shell as a simple command. Any operands passed to the function can be accessed within the function using the positional parameters. The special parameter **\$#** is also temporarily overridden to be the number of operands passed to the function. The body of the function is the *list* of commands.

Regular built-in utilities

These “regular” utilities may be found external to the Shell, but they have been built into the Shell to enhance system performance.

The following regular built-in utilities are briefly described here. For a more complete description of **cd**, **kill**, **onnode**, **ontty**, and **umask**, see the *Utilities Reference*.

cd change working directory

cd [*directory*]

The **cd** utility changes the working directory of the current execution environment. If *directory* isn’t specified, the value of the **HOME** environment variable becomes the new working directory.

command execute a simple command

command [-p] *command* [*argument...*]

The **command** utility causes the Shell to treat the *arguments* as a simple command, suppressing the shell function lookup that is part of the default command lookup process. If the **-p** option is specified, the command search is performed using a default value for the **PATH** environment variable, which looks in the standard places for the utilities.

false return a false value

false

The **false** utility returns a nonzero exit status.

getopts parse utility options

getopts *optstring name* [*argument...*]

The **getopts** utility uses the characters in *optstring* to parse and retrieve options and option arguments from a list of parameters. The *optstring* argument is a string containing the option characters to be recognized. A colon following an option character informs **getopts** that this option is expected to have an option argument.

By default, **getopts** parses the positional parameters passed to the invoking shell procedure. If one or more arguments are given, the arguments are parsed instead of the positional parameters.

Each time it's invoked, the **getopts** utility places the value of the next option in the shell variable specified by *name*; it also places the index of the next argument to be processed in the shell variable **OPTIND**. When the Shell is first invoked, **OPTIND** is initialized to 1. When an option requires an option argument, it is stored in the variable **OPTARG**. If no option is found, or if an option is found but it doesn't have an option argument, **OPTARG** is unset.

If an option character not contained in *optstring* is found, *name* is set to the question mark character (?), **OPTARG** is unset, and an error message is written to the standard error output. Under this condition, if the first character of *optstring* is a colon, **OPTARG** is instead set to the option character found and no error message is output.

When the **getopts** utility has processed all of the options, it returns with an exit status greater than zero and the **OPTIND** variable is set to the index of the first nonoption argument.

kill terminate or signal processes

kill -s *signame pid...*

kill -*signum pid...*

kill -l

The **kill** utility sends the signal specified by *signame* or *signum* to the specified processes (if a *pid* is negative, it specifies a process group). In the first form, *signame* is the symbolic name of a signal

without the SIG prefix (e.g. QUIT or HUP). In the second form, *signum* is the decimal number of the signal to be set. If no signal is specified, the termination signal is set by default.

If the **-1** (el) option is specified, the **kill** utility writes a list of all supported signal names to the standard output.

onnode start a process on another node

onnode *node command* [*argument...*]

The **onnode** utility executes the command with the specified arguments on the specified node. The standard input, standard output, and standard error output are inherited from the invoking shell.

The syntax:

```
onnode 1 sin
```

is the same as

```
//1 sin
```

ontty start a process on another tty

ontty *device command* [*argument...*]

The **ontty** utility executes the command with the specified arguments on the specified tty. The command is detached from the invoking terminal.

This command can be combined with the remote process creation syntax.

In the following example, **mymonitor** is run on the local node, with its input and output attached to **/dev/ser1** of node 4:

```
ontty //4/dev/ser1 mymonitor
```

The following example would attach and run **mymonitor** on **/dev/ser1** of node 4 (**mymonitor** would run on node 4 with no virtual circuits back to the node it was started from):

```
onnode 4 ontty //4/dev/ser1 mymonitor
```

read read a line from standard input

```
read [-r] var...
```

The **read** utility reads a single line from the standard input. The line is split into words using the separator characters contained in the **IFS** variable. The first word is assigned to the first *var*, the second word to the second *var*, and so forth. Any additional words are appended to the last *var*. If the number of *var* arguments specified is greater than the number of words in the input line, the extra *var* arguments are assigned the null string.

If the **-r** option is specified, the backslash character (\) is treated like any other character in the input line instead of being interpreted as an escape character.

true return true value

```
true
```

The **true** utility returns a zero exit status.

umask get or set the file mode creation mask

```
umask [-s] [mask]
```

The **umask** utility sets the file mode creation mask of the current shell environment to the value specified by the *mask* operand. The *mask* operand can be either an octal value or a symbolic operand. If the *mask* operand isn't specified, the **umask** utility writes the current value of the file mode creation mask to the standard output. If the **-s** option is specified the **umask** utility writes symbolic output.

wait await process creation

wait [*pid...*]

The **wait** utility waits for the specified processes to terminate and returns the exit status of the process requested by the last *pid* operand. If no operands are specified, it waits for all of the child processes of the invoking shell to terminate.

Special built-in commands

The following special commands are “built in” to the Shell. That is, the Shell interprets and executes them internally. Because these commands are tightly bound to the shell interface, they are called *special* built-in commands. The output of these commands, if any, is written to the standard output; you can apply normal redirection and piping operations.

break exit from a **for**, **while**, or **until** loop

break [*n*]

The **break** command exits from the smallest enclosing **for**, **while**, or **until** loop, or from the *n*th enclosing loop if *n* is specified (default for *n* is 1). If *n* is greater than the number of enclosing loops, then the largest enclosing loop is exited. Execution continues with the command immediately following the exited loop.

: (colon) null command

:

The **:** (colon) command does nothing; it is simply a null command or place holder. It always returns an exit status of zero.

continue continue a **for**, **while**, or **until** loop

continue [*n*]

The **continue** command returns to the top of the smallest enclosing **for**, **while**, or **until** loop, or to the top of the *n*th enclosing loop if *n* is specified (default for *n* is 1). If *n* is greater than the number of enclosing loops, then execution resumes at the top of the largest enclosing loop.

. (dot) execute commands in the current environment

. *file*

The **.** (dot) command reads and executes the commands from *file* within the current environment. The search path contained in the environment variable **PATH** is used to find the directory containing *file*. This command doesn't require that *file* be executable.

eval execute a command from concatenated arguments

eval [*argument...*]

The **eval** command constructs a command by concatenating the specified arguments. The constructed command is then read as input to the Shell and executed.

exec execute a command and/or manipulate file descriptors

exec [*command* [*argument...*]]

The **exec** command opens, closes, and/or copies file descriptors as specified by any I/O redirections given as part of *argument*. If a command is specified, that command is spawned as a replacement for the Shell. Any specified arguments are passed to the spawned process.

exit cause the Shell to exit

exit [*n*]

The **exit** command causes the Shell to exit with the exit status specified by *n*. If *n* isn't specified, the exit status is that of the last command executed.

export mark environment variables for export

export *name* [=word] ...

export -p

The **export** command marks the specified names for **export**, which will cause them to be in the environment of subsequently executed commands. If you specify the **-p** option, the names and values of all exported variables are written to the standard output.

readonly mark environment variables as read-only

readonly *name* [=word] ...

readonly -p

The **readonly** command marks the specified names as “read-only.” Environment variables marked with the **readonly** attribute cannot be changed by a subsequent assignment nor can they be unset by the **unset** command.

If the **-p** option is given, the names and values of all **readonly** variables are written to the standard output.

return return from a function

return [*n*]

The **return** command causes the Shell to exit the current function or **dot** script. The return value is either the value specified by *n* or the exit status of the last command executed if *n* isn't specified.

set set/unset options and positional parameters

set [-Caefnuvx] [argument...]

set [+Caefnuvx] [argument...]

set -- [argument...]

The **set** command sets and/or unsets the shell options. An option preceded by:

- sets the corresponding attribute
- + unsets the corresponding attribute

The current set of option flags is kept in the special parameter -. .



If any *arguments* are given, the positional parameters are overridden with these values and the special parameter \$# is updated. Note also that the double dash (--) is used to separate the options from the *arguments*.

- C Don't overwrite existing files. This is the *noclobber* option, which prevents the Shell's > redirection operator from overwriting existing files. You can use the >| redirection operator to override this option for individual files.
- a Set the export attribute for each variable that is created or assigned a new value.
- e Exit immediately if a command returns with a nonzero exit status.
- f Disable filename expansion.
- n Read commands but don't execute them. This option can be used to check a shell script for syntax errors before actually executing it.
- u Treat the expansion of unset named parameters as an error.
- v Be verbose; write input lines to standard error before execution.
- x Enable execution trace; write each command and its arguments to the standard error output before execution.

shift shift positional parameters

shift [*n*]

The **shift** command shifts positional parameters down by *n* positions. If *n* isn't specified, it defaults to 1. Positional parameter 1 becomes parameter (1+*n*), parameter 2 becomes parameter (2+*n*), and so forth. The special parameter **\$#** is updated to contain the new number of positional parameters.

trap trap signals

trap [*action condition...*]

The **trap** command causes the specified *action* to be executed when one of the specified *conditions* occurs. A *condition* is either a signal specified using a symbolic name, with or without the SIG prefix, or the string EXIT. The *action* is simply a Shell command and is processed by the Shell in a manner equivalent to:

eval "\$*action*"

Signals that were ignored on entry to a non-interactive shell cannot be trapped or reset, whereas an interactive shell may reset or catch them. Traps remain in place for a given shell until they are explicitly changed with another **trap** command.

If no arguments are given, the **trap** command writes a list of *actions* associated with each *condition* to the standard output.

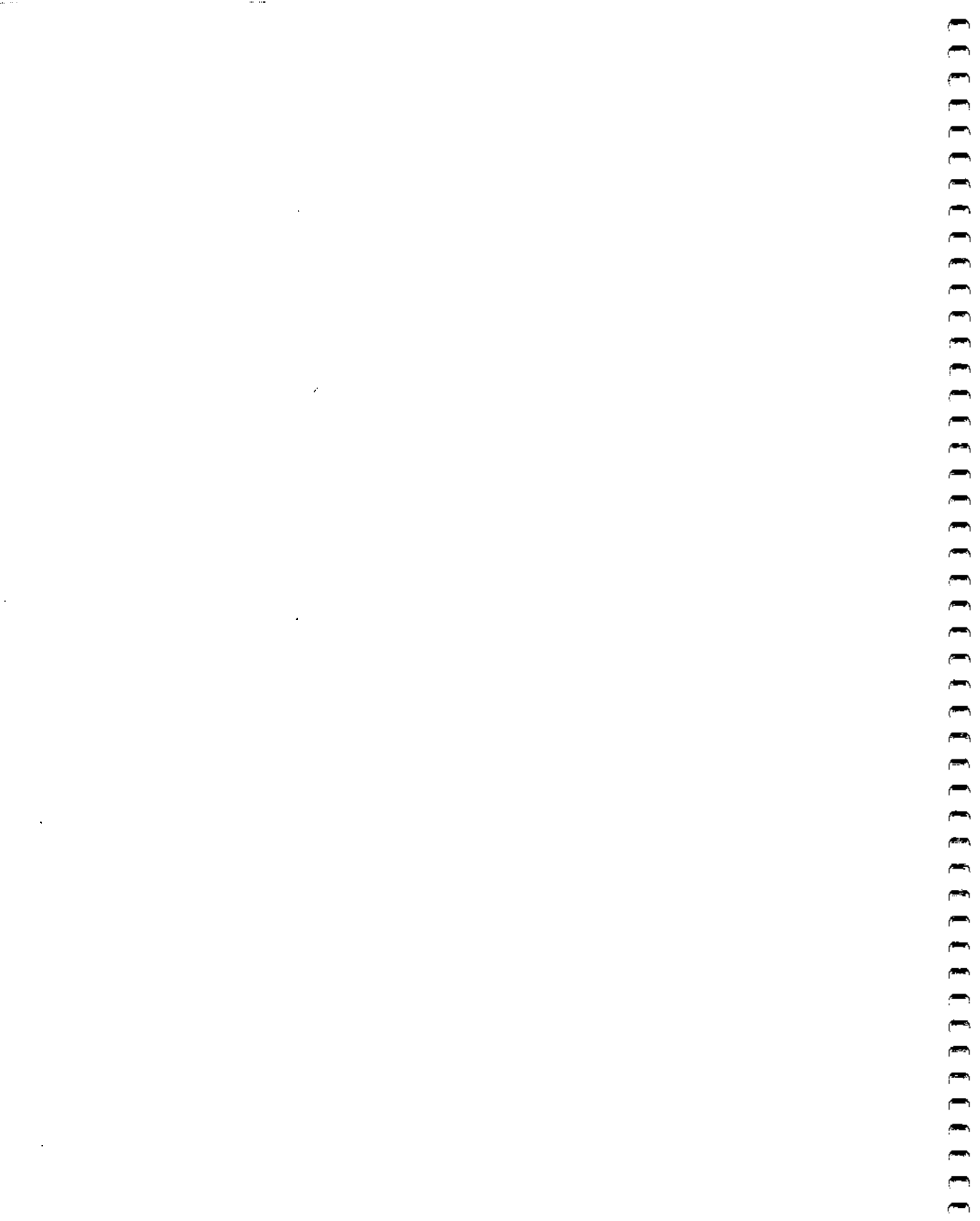
unset unset values and attributes of named parameters and functions

unset *name...*

unset **-f** *name...*

The **unset** command unsets each named parameter or function specified. If the **-f** option is specified, then the function *name* is unset. If the **-f** option is **not** specified, then *name* refers to a named parameter. If that named parameter doesn't exist, then the function of that name, if any, is unset.

You can't use the **unset** command to unset read-only named parameters, positional parameters, and special parameters.





Appendix A

The QNX Terminal

In this Appendix...

Standard console character set

QNX keyboard codes

Composed character input sequences

Terminal escape sequences



Introduction

QNX supports a logical terminal on the virtual consoles. That is, it can recognize certain escape sequences and special characters and can perform terminal control functions when the special sequences are encountered. A number of manufacturers have also developed real ASCII terminals that use the same protocol as the QNX logical terminal. This terminal is defined in the **terminfo** database as the **qnx** terminal.

Standard console character set

DECIMAL VALUE	➡	0	16	32	48	64	80	96	112
↩	HEXA DECIMAL VALUE	0	1	2	3	4	5	6	7
0	0	BLANK (NULL)	▶	BLANK (SPACE)	0	@	P	‘	p
1	1	😊	◀	!	1	A	Q	a	q
2	2	😄	↕	"	2	B	R	b	r
3	3	♥	!!	#	3	C	S	c	s
4	4	♦	⌘	\$	4	D	T	d	t
5	5	♣	§	%	5	E	U	e	u
6	6	♠	▬	&	6	F	V	f	v
7	7	•	↕	'	7	G	W	g	w
8	8	•	↑	(	8	H	X	h	x
9	9	○	↓	)	9	I	Y	i	y
10	A	●	→	*	:	J	Z	j	z
11	B	♂	←	+	;	K	l	k	{
12	C	♀	└	,	<	L	\	l	
13	D	🎵	↔	—	=	M	l	m	}
14	E	🎵	▲	.	>	N	^	n	~
15	F	☀	▼	/	?	O	—	o	△

DECIMAL VALUE	◆	128	144	160	176	192	208	224	240
◆	HEXA DECIMAL VALUE	8	9	A	B	C	D	E	F
0	0	Ç	É	á	▤	▤	▤	∞	≡
1	1	ü	æ	í	▤	▤	▤	β	±
2	2	é	Æ	ó	▤	▤	▤	Γ	≥
3	3	â	ô	ú	▤	▤	▤	π	≤
4	4	ä	ö	ñ	▤	▤	▤	Σ	∫
5	5	à	ò	Ñ	▤	▤	▤	σ	∫
6	6	å	û	ä	▤	▤	▤	μ	÷
7	7	ç	ù	ó	▤	▤	▤	τ	≈
8	8	ê	ÿ	ì	▤	▤	▤	ø	°
9	9	ë	Ö	▤	▤	▤	▤	θ	•
10	A	è	Ü	▤	▤	▤	▤	Ω	•
11	B	ï	£	½	▤	▤	▤	δ	√
12	C	î	ℒ	¼	▤	▤	▤	∞	n
13	D	ì	¥	ì	▤	▤	▤	φ	²
14	E	Ä	℞	«	▤	▤	▤	€	■
15	F	Å	ƒ	»	▤	▤	▤	∩	BLANK 'FF'

QNX keyboard codes

		Upper 4 bits							
		0	1	2	3	4	5	6	7
Normal codes	0	NUL	DLE	SP	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

Control codes

NUL - Ctrl @	BS - Ctrl h	DLE - Ctrl p	CAN - Ctrl x
SOH - Ctrl a	HT - TAB	DC1 - Ctrl q	EM - Ctrl y
STX - Ctrl b	LF - Ctrl j	DC2 - Ctrl r	SUB - Ctrl z
ETX - Ctrl c	VT - Ctrl k	DC3 - Ctrl s	ESC - Esc
EOT - Ctrl d	FF - Ctrl l	DC4 - Ctrl t	FS - Ctrl \
ENQ - Ctrl e	CR - Enter	NAK - Ctrl u	GS - Ctrl]
ACK - Ctrl f	SO - Ctrl n	SYN - Ctrl v	RS - Ctrl ^
BEL - Ctrl g	SI - Ctrl o	ETB - Ctrl w	US - Ctrl _

Function codes

Upper 4 bits

	0	...	8	9	A	B	C	D	E	F	
0			Shift Tab	Shift F6	Home	Ctrl Home	Alt Home	Ctrl Enter	Shift Enter	Alt p	0
1	Key Up		F1	Shift F7	Up	Ctrl Up	Alt Up	Alt F1	Alt a	Alt q	1
2	Shift Down		F2	Shift F8	PgUp	Ctrl PgUp	Alt PgUp	Alt F2	Alt b	Alt r	2
3	Shift Up		F3	Shift F9	Minus	Ctrl Minus	Alt Minus	Alt F3	Alt c	Alt s	3
4	Ctrl Down		F4	Shift F10	LEFT	Ctrl LEFT	Alt LEFT	Alt F4	Alt d	Alt t	4
5	Ctrl Up		F5	Ctrl F1	Five	Ctrl Five	Alt Five	Alt F5	Alt e	Alt u	5
6	Alt Down		F6	Ctrl F2	RIGHT	Ctrl RIGHT	Alt RIGHT	Alt F6	Alt f	Alt v	6
7	Alt Up		F7	Ctrl F3	Plus	Ctrl Plus	Alt Plus	Alt F7	Alt g	Alt w	7
8	Extend Key		F8	Ctrl F4	End	Ctrl End	Alt End	Alt F8	Alt h	Alt x	8
9			F9	Ctrl F5	DOWN	Ctrl DOWN	Alt DOWN	Alt F9	Alt i	Alt y	9
A			F10	Ctrl F6	Pgdn	Ctrl Pgdn	Alt Pgdn	Alt F10	Alt j	Alt z	A
B			Shift F1	Ctrl F7	Ins	Ctrl Ins	Alt Ins	Shift F11	Alt k		B
C			Shift F2	Ctrl F8	Del	Ctrl Del	Alt Del	Shift F12	Alt l		C
D			Shift F3	Ctrl F9	PrtSc	Ctrl PrtSc	Alt *	SysRq	Alt m		D
E			Shift F4	Ctrl F10	F11	Ctrl F11	Alt F11	Ctrl BS	Alt n		E
F			Shift F5	Ctrl Tab	F12	Ctrl F12	Alt F12		Alt o		F
	0	...	8	9	A	B	C	D	E	F	

Note The above codes are *always* preceded by a hex FF code.

Codes in the range FF 00 to FF 0F are generated only when the console is in SCAN mode.

Composed character input sequences

Alt C ,	Ç	Alt E ´	É	Alt a ´	á	Alt g a	α
Alt u ¨	ü	Alt A e	æ	Alt i ´	í	Alt g b	β
Alt e ´	é	Alt A E	Æ	Alt o ´	ó	Alt g g	Γ
Alt a ^	â	Alt o ^	ô	Alt u ´	ú	Alt p i	π
Alt a ¨	ä	Alt o ¨	ö	Alt n ~	ñ	Alt g e	Σ
Alt a ´	à	Alt o ´	ò	Alt N ~	Ñ	Alt g s	σ
Alt a o	å	Alt u ^	û	Alt a _	ä	Alt g u	μ
Alt c ,	ç	Alt u ´	ù	Alt o _	ó	Alt g t	τ
Alt e ^	ê	Alt y ¨	ÿ	Alt ? ?	¿	Alt o l	Φ
Alt e ¨	ë	Alt O ¨	Ö	Alt + _	⌋	Alt o -	θ
Alt e ´	è	Alt U ¨	Ü	Alt _ +	⌈	Alt g w	Ω
Alt i ¨	ï	Alt c l	¢	Alt / 2	½	Alt g d	δ
Alt i ^	î	Alt L -	£	Alt / 4	¼	Alt o o	∞
Alt i ´	ì	Alt Y =	¥	Alt ! !	¡	Alt o /	φ
Alt A ¨	Ä	Alt P t	Pₜ	Alt < <	«	Alt E E	€
Alt A o	Å	Alt f -	f	Alt > >	»	Alt U U	∩
Alt = =	≡						
Alt + -	±						
Alt > =	≥						
Alt < =	≤						
Alt - :	÷						
Alt ~ ~	≈						
Alt o .	•						
Alt . .	•						
Alt s q	√						
Alt g n	n						
Alt ^ 2	²						

Note To generate any of these characters, you:

- 1) press and release <Alt>
- 2) type the indicated two-character sequence

Terminal escape sequences

The following escape sequences and special characters cause a **qnx** terminal to perform specified control functions. Escape sequences are signalled by an escape character **<Esc>** (1B hex) followed by a set of ASCII characters. Any other character preceded by an **<Esc>** character, including **<Esc>**, loses its significance as a control function and is simply displayed as a graphic character on the screen.

In the following table, spaces are included in the escape sequences to make the sequences easier to read—these spaces don't actually form part of the sequences.

	If you want to:	you use:
Setting character attributes	turn on inverse video mode (if a terminal supports color, this also swaps the foreground and background colors set by the <Esc> @ sequence)	<Esc> (
	turn off inverse video	<Esc>)
	turn on underline mode (available only on monochrome)	<Esc> [
	turn off underline mode	<Esc>]
	turn on blinking mode	<Esc> {
	turn off blinking mode	<Esc> }
	turn on high-intensity mode	<Esc> <
	turn off high-intensity mode	<Esc> >
Defining and moving the cursor	define the type of cursor, where <i>c</i> is the cursor type (specify ASCII character 0 for no cursor, 1 for a normal cursor, and 2 for a block cursor)	<Esc> y c
	position the cursor at the screen coordinates <i>r</i> and <i>c</i> , where <i>r</i> is the row and <i>c</i> is the column; both <i>r</i> and <i>c</i> are offset by 20 hex (a space). Therefore, <i>rc</i> = <i><space><space></i> is row zero, column zero	<Esc> = r c or <Esc> Y r c
	move cursor up one line without wrap or scrolling	<Esc> A
	move cursor down one line without wrap or scrolling	<Esc> B

	If you want to:	you use:
Defining and moving the cursor (cont'd)	move the cursor down one line (if the cursor is positioned on the last line of the screen, the screen scrolls up one line)	LF (0A hex)
	move cursor right one column without wrap	<Esc> C
	move cursor left one column without wrap	<Esc> D
	move cursor to the top left corner of the screen	<Esc> H
	turn auto line wrapping on	<Esc> h
	turn auto line wrapping off	<Esc> i
	move cursor up with wrap	<Esc> a
	move cursor down with wrap	<Esc> b
	move cursor right with wrap	<Esc> c
	move cursor left with wrap	<Esc> d
	reverse linefeed (the screen scrolls down when the cursor is positioned on the top line of the screen)	<Esc> I
	move the cursor to the first column of the current row	CR (0D hex)
Inserting, deleting and clearing	enter a nondestructive backspace (wraps to the end of the previous line when at the beginning of the line; wraps to the bottom right corner of the screen when at the top left corner of screen)	BS (08 hex)
	insert line at cursor location	<Esc> E
	delete line at cursor location	<Esc> F
	clear to end of screen with spaces (uses the fill color specified with the <Esc> ! sequence on color terminals)	<Esc> J
	clear to end of line with spaces (uses the fill color specified with <Esc> ! on color terminals)	<Esc> K

	If you want to:	you use:
	clear the entire screen to spaces (uses the fill color specified with <Esc> ! and moves the cursor to the top left corner)	FF (0C hex)
	insert space at cursor position	<Esc> e
	delete character at cursor position	<Esc> f
	repeat the character <i>c</i> the number of times specified by <i>n</i> , where <i>n</i> is offset by 20 hex	<Esc> g n c
Saving attributes	save color and attribute information (this overrides any previously saved information; the cursor position isn't saved. QNX programs, such as editors that change the colors, assume that a default has been set on entry and use <Esc> R to restore the default on exit)	<Esc> S
	restore the color and attribute information saved with <Esc> S (the cursor position isn't affected)	<Esc> R
Setting color	define the foreground and background color of displayed characters, where <i>fc</i> is the color of the cursor, and <i>bc</i> is the fill color—both <i>fc</i> and <i>bc</i> represent a single character taken from the color table below. This has no effect on a monochrome terminal or display.	<Esc> @ fc bc
	set the fill color that is used both for blank lines created by scrolling and for the entire screen when clearing with a form feed. This has no effect on a monochrome terminal or display.	<Esc> ! fc bc
Generating a tone	generate a short tone on the system console or ring the bell on a terminal	BELL (07 hex)

Color codes

You can use these color codes in the above escape sequences:

Color	ASCII character
black	0
blue	1
green	2
cyan	3
red	4
magenta	5
brown	6
white	7



A

- adaptive scheduling 38
- alias prefixes 60 – 62
- aliases 60 – 62
- .altboot** file 83
- appending output to a file 113, 137
- architecture of QNX 5 – 6
- arithmetic expansion (Shell) 128
- arithmetic operations, signal generated by
 - erroneous operation 26
- arming timers 54
- arrow keys 102 – 103

B

- Backspace key 102
- bad blocks on disk 77
- bell, ringing on a terminal 167
- bitmap 83
- block special files 68, 78 – 81
 - defining 80
 - defining entire disk as 78
 - prefix tree where names are placed 80
 - registering prefixes 80
- blocked states
 - see* process states

- blocking signals 29
- blocks
 - bad blocks on disk 77
 - loader block 82
 - numbering 78
 - root block 83
 - used blocks on disk 83
- booting 106
 - .altboot** file 83
 - .boot** file 83
- break** (Shell built-in) 150
- breaks, recognizing and acting on 86
- buffer cache (Filesystem Manager) 75

C

- C functions
 - _exit()** 29
 - abort()** 26
 - abstimer()** 55
 - access()** 29
 - alarm()** 26, 29
 - cfgetispeed()** 29
 - cfgetospeed()** 29
 - cfsetispeed()** 29
 - cfsetospeed()** 29
 - chdir()** 29, 60

chmod() 29
chown() 29
close() 29, 78, 86
closedir() 70
console_active() 94
console_arm() 94
console_open() 94
console_read() 94
console_size() 94
console_write() 94
creat() 29
Creceive() 21
Creceivemx() 21
dev_arm() 87
dev_insert_chars() 87
dev_read() 87
dup() 29, 65
dup2() 29, 65
exec() 46 – 47
execle() 29
execve() 29
exit() 48
fcntl() 29, 65
fcntlrl() 66
fork() 29, 46 – 47
fstat() 29
getegid() 29
geteuid() 29
getgid() 29
getgroups() 29
getpgrp() 29
getpid() 29
getppid() 29
getprio() 36
getscheduler() 37
gettimer() 55
getuid() 29
kill() 24, 29
link() 29, 70 – 71
longjmp() 28
lseek() 29, 64 – 65
mkdir() 29
mkfifo() 29, 74
mktimer() 54
open() 29, 33, 58 – 59, 61, 63 – 64, 70, 78, 86 – 87
opendir() 70
pathconf() 29
pause() 29
pipe() 29, 74
popen() 74
qnx_fullpath() 63
qnx_hint_attach() 56
qnx_hint_detach() 56
qnx_ioctl() 92
qnx_name_attach() 53
qnx_name_locate() 33, 53
qnx_osinfo() 52
qnx_pflags() 27, 40
qnx_proxy_attach() 23
qnx_psinfo() 52
qnx_spawn() 49
qnx_spawn_options() 49
qnx_timerperiod() 55
qnx_vc_attach() 31
raise() 24
read() 29, 70, 78, 86
readdir() 70
Readmsg() 22
Readmsgmx() 21
Receive() 13, 16
Receivemx() 21
reltimer() 55
rename() 29
Reply() 13, 16, 18
Replymx() 21
rewinddir() 70
rmdir() 29
rmtimer() 55
Send() 13, 16

- Sendfd()* 63
- Sendmx()* 21
- setgid()* 29
- setpgid()* 29
- setprio()* 36
- setscheduler()* 37
- setsid()* 29
- setuid()* 29
- sigaction()* 28 – 29
- sigaddset()* 29
- sigdelset()* 29
- sigemptyset()* 29
- sigfillset()* 29
- sigismember()* 29
- siglongjmp()* 28
- signal()* 28 – 29
- sigpending()* 29
- sigprocmask()* 29
- sigsuspend()* 29
- sleep()* 29, 53
- sleepms()* 53
- spawn()* 46 – 47, 62
- stat()* 29
- sysconf()* 29
- tcdrain()* 29, 87
- tcdropline()* 87
- tcflow()* 29
- tcflush()* 29
- tcgetattr()* 29, 86
- tcgetpgrp()* 29
- tcsendbreak()* 87
- tcsetattr()* 29, 86
- tcsetgrp()* 29
- time()* 29
- times()* 29
- Trigger()* 23
- tscendbreak()* 29
- umask()* 29
- uname()* 29
- unlink()* 29, 72, 74
- ustat()* 29
- utime()* 29
- wait()* 29, 49 – 50
- waitpid()* 29, 49 – 50
- write()* 29, 78, 86
- Writemsg()* 22
- Writemsgmx()* 21
- case** conditional construct (Shell) 144
- catching signals 28
- cd** (Shell built-in) 146
- CDPATH (environment variable) 127
- character special files 68
- characters
 - changing size of displayed characters 94, 105
 - composed character input sequences 164
 - console character set 160
 - deleting 88, 102, 166
 - echoing 86
 - inserting 88, 102
 - international characters 97, 160, 164
 - mathematical characters 160, 164
 - overwriting 88, 102
 - pattern-matching characters 114
 - setting character attributes 165
 - translation of output characters 86
- client-driven scheduling 39
- client/server model 18, 39
- colon** (Shell built-in) 150
- color, setting (**gnx** terminal) 167
- command** (Shell built-in) 146
- command interpreter
 - see* Shell
- command lines, entering 112
- command lists 126
- command output
 - see* output
- command-line recall 103
- commands
 - basics 109
 - command substitution 127

- common file management commands 122
- displaying usage messages 113
- entering 112
- key symbols 111
- multiple commands, entering 126
- recalling 103
- shortcuts 114
- structure 110
- substituting a command with its stdout 127
- see also* utilities
- communications channel
 - disconnecting 87
 - sending breaks across 87
 - serial channel management 95
- composed character input sequences 164
- conditional message reception 21
- console character set 160
- consoles
 - console-specific C functions 94
 - controlling console size 94
 - controlling visible console 94
 - defined 92 – 94
 - generating a tone 167
 - logging in on an unused console 105
 - multiple concurrent sessions 92 – 94
 - multiple consoles, using 104
 - notification of significant events 94
 - reading screen 94
 - screen size, changing 94, 105
 - terminating sessions 105
 - writing directly to screen 94
 - see also* virtual consoles
- continue** (Shell built-in) 151
- control operators (Shell) 142
- controlling process
 - signal generated upon death 26
- controlling terminal
 - signal generated upon hangup 26
- cooperating processes 13
 - benefits of 52

- FIFOs 74
- pipes 74
- residing on different nodes 31
- special considerations 52
- synchronizing via message passing 15
- see also* processes
- copying files 122
- creating
 - directories 122
 - processes 46, 48
- Creceive()* (message-passing primitive) 22
- current working directory 72, 119
 - changing 122
 - printing 122
 - see also* directories
- cursor
 - defining 165
 - moving 88, 102, 165

D

- data flow
 - between applications and devices 86
 - between Dev and its drivers 91
- date and time stamps 68
- DEAD (process state) 50
- Del key 102
- deleting
 - characters 88, 102, 166
 - directories 122
 - files 122
- Dev.con** (driver process) 92 – 94
- Dev.par** (driver process) 96
- Dev.ser** (driver process) 95
- device control 92
 - low-level device control 92
- device drivers
 - see* drivers

Device Manager
 auxiliary services for terminal devices 87
 general responsibilities 86
 device names, creating 61
 devices
 accessing across the network 59
 device subsystem performance 97
 parallel devices 96
 serial devices 95
 see also terminal devices
 directories 68 – 69
 changing 122, 146
 commonly used commands 122
 creating 122
 current working directory 72, 119, 122
 directory entries 69
 directory entries, extent information 72
 directory entries, reading 70
 directory links 72
 dot (link to current working directory) 72
 dot dot (link to parent directory) 72
 extents 72
 filenames 69 – 70
 home directory 120
 home directory, returning to 122
 inodes 70
 linking 70
 links 70
 listing contents 122
 navigating directory structure 69, 119
 parent directory 72
 removing 122
 root directory, special characteristics 83
 writing 70
 see also files
 disconnecting communications channel 87
 disk subsystems 78
 components 78
 multiple 78

disks
 bad blocks on disk 77
 how blocks are numbered 78
 multiple operating systems on a disk 78
 partition types 78
 ramdisks 76
 used blocks 83
 viewed as block special files 78
 displayed characters
 changing size 94, 105
 dot (link to current working directory) 72
 dot (Shell built-in) 151
 dot dot (link to parent directory) 72
 down arrow key 103
 driver processes
 Dev.con 92 – 94
 Dev.par 96
 Dev.ser 95
 drivers 7, 90 – 91
 effects of adding new drivers 7
 dumper (utility) 49
 duplicating file descriptors 65

E

echoing characters 86
 edited input mode 87 – 88
 elevator seeking (Filesystem Manager) 75
 End key 102
 Enter key 102
 entering
 input 102
 multiple commands 115
 text vii
 environment variables 127
 marking as read-only 152
 marking for export 152
 see also named parameters
 eval (Shell built-in) 151

- event-reporting mechanisms
 - notify with proxy 54
 - notify with signal 54
 - sleep until completion 54
 - exec** (Shell built-in) 151
 - exec()* (process creation primitive) 46
 - executing processes 48
 - see also* processes
 - exit** (Shell built-in) 152
 - export** (Shell built-in) 152
 - extending files 73
 - extensibility of QNX 6
 - extents 72
 - allocating new extents 73
 - benefits of separating extents from data 72
 - how files are extended 73
 - where extents are stored 72
- ## F
- false** (Shell built-in) 146
 - FIFO scheduling 37
 - FIFOs 68, 74
 - creating 74
 - deleting 74
 - see also* pipes
 - file descriptors
 - closing 139
 - duplicating 65, 138
 - inheritance 65
 - manipulating 151
 - namespace 63 – 66
 - open control blocks 63
 - opening 138
 - preventing inheritance 66
 - several FDs referring to the same OCB 65
 - file mode creation mask 149
 - filenames
 - defined 69
 - dot (link to current working directory) 72
 - dot dot (link to parent directory) 72
 - expansion 129
 - longer than 16 characters 71
 - maximum length 118
 - portable 118
 - suffixes 118
 - files
 - accessing across the network 59, 119
 - appending 113
 - block special files 68
 - character special files 68
 - closing 71
 - combining 122
 - commonly used commands 122
 - comparing 122
 - copying 122
 - counting words, lines, or bytes 122
 - date and time stamps 68
 - defined 68
 - directories 68 – 69, 119
 - displaying contents 122
 - extending 73
 - extents 72
 - FIFOs 68, 74
 - filenames 69
 - links 70 – 71
 - locating 69
 - moving 122
 - multiple filenames 70 – 71
 - multiple files, working with 114, 129
 - naming 118
 - opened by different processes 64
 - opened twice by same process 64
 - pipes 68, 74
 - printing 122
 - recovering 78
 - regular files 68 – 69
 - removing 122
 - renaming 122

- searching for string patterns 122
- sorting 122
- times maintained for each file 68
- types of files implemented in QNX 68
- see also* directories; extents; filenames
- filesystem
 - accessing a filesystem on another node 60
 - implemented on block special files 69
 - mounting 81
 - recovering 77
 - robustness 77
 - seek points 64
 - typical filesystem 120 – 121
- Filesystem Manager
 - accessing several devices in parallel 76
 - buffer cache 75
 - client-driven priority 76
 - elevator seeking 75
 - filetypes managed by **Fsys** 68
 - multi-threading 76
 - multiple I/O requests, managing 76
 - performance 75 – 76
 - ramdisk 76
 - responsibilities 68
 - satisfying I/O requests from buffer cache
 - while other requests in progress 76
 - temporary files 76
 - see also* files; filesystem; read operations; write operations
- flow control 86
- flow of data between:
 - applications and devices 86
 - Dev** and its drivers 91
- for** loop (Shell) 145
- fork()** (process creation primitive) 46
- function definition command (Shell) 145

G

- getopts** (Shell built-in) 147
- global names 53

H

- hangups
 - recognizing and acting on 86
- hardware instructions, signal generated upon
 - detection of invalid instruction 26
- hardware interrupts
 - see* interrupts
- HOME (environment variable) 127
- home directory 120
 - returning to 122
- Home key 102

I

- I/O namespace 58
 - file descriptor namespace 63 – 66
 - prefix aliases 60 – 62
 - resolving pathnames 58 – 59
- I/O servers
 - interrupt-driven 95
 - non-interrupt servers 96
- I/O services 58
 - file-oriented 58
- if** conditional construct (Shell) 144
- IFS (environment variable) 128
- images 83
- inodes
 - .inodes** file 70
 - where inode information is stored 70
- input
 - deleting 88, 102, 166
 - edited input mode 87 – 88

- entering 102
- line-oriented input 102
- lines, editing 86, 102
- lines, recalling 103
- maximum length of input line 103
- model 97
- raw input mode 89
- reading from a file 113
- redirecting 74, 113, 137 – 138
- submitting line of input 102

Ins key 102

inserting

- characters 88, 102

international characters

- entering 97, 160, 164

international keyboards 97 – 98

interprocess communication (IPC) 7, 13

- across the network 31 – 34
- via messages 13 – 22
- via proxies 23
- via signals 24 – 30
- see also* message passing; proxies; signals; virtual circuits; virtual processes

interrupt handlers 41, 55 – 56

- placing received data into memory queue 97
- timer interrupt handlers 56
- see also* proxies

interrupt latency 40

interrupt-driven I/O servers 95

- see also* non-interrupt I/O servers

interrupts 40

- priorities 42
- stacked 42

inverse video mode 165

IOT instruction, signal generated 27

IPC

- see* interprocess communication

K

kernel

- entered as result of kernel calls 6
- general responsibilities 5, 12
- scheduling processes 35 – 39
- size 5

keyboard

- conventions used in this manual vii
- international keyboards 97 – 98
- key chords vii
- summary of functions 107
- using 101, 103, 105, 107

keyboard codes (**qnx** terminal) 162

kill (Shell built-in) 24, 147

L

LAN

- see* local area networking
- see also* network

latency

- interrupt latency 40
- scheduling latency 41

left arrow key 102

line editing 86

line-control discipline 86

line-oriented input

- entering 102

links 70

- creating 71
- directory 72
- limitations 70
- removing 71 – 72

loader block 82

loading processes 48

local area networking 8 – 10

local names 53

locating
 files 69, 119
 processes 52
 logging in on an
 unused console 105
 LOGNAME (environment variable) 128

M

managers (system processes) 6
 similarity to user-written processes 6
 see also Device Manager; Filesystem Manager;
 Network Manager; Process Manager
 mathematical characters
 entering 160, 164
 memory parity error, signal generated 27
 memory references, signal generated upon
 detection of invalid reference 26
 memory-image dump, generating when
 a process is terminated by a signal 49
 message passing 7, 13 – 22
 advanced message-passing facilities 20
 conditional message reception 21
 invoking messages 13
 message data, where stored 18
 messages containing variable amounts
 of data 22
 messages defined 13
 messages, contents of 8, 13
 messages, purpose of 13
 multi-part messages 20
 order in which messages are received 19
 primitives 13, 22
 process synchronization 15
 reading part of a message 22
 reserved message codes 22
 signals and messages 30
 using buffer space already allocated for
 a message 22

 writing part of a message 22
 zero-length messages 19
 zero-length replies 19
 see also interprocess communication; proxies;
 signals; virtual circuits
 microkernel architecture 5 – 6
 modem
 connected to a serial port 88
 mounting filesystems 81
 moving
 cursor 88, 102, 165
 files 122
 multi-part messages 20
 multi-sector writes 75
 multi-threading (Filesystem Manager) 76
 multiple commands, entering 115

N

named parameters (Shell) 131
 unsetting values and attributes 154
 see also environment variables
 names
 see filenames
 see also process names
 network
 as homogeneous set of resources 8 – 10
 executing commands on remote nodes 139
 flexibility 9
 integrity 33
 nodes leaving the network 33
 pathnames 119
 root 59
 transparency 8 – 9, 31
 virtual processes 31
 Network Manager
 virtual circuits 31
 network-wide IPC 31 – 34
 non-interrupt I/O servers 96

notify with proxy
 (event-reporting mechanism) 54
notify with signal
 (event-reporting mechanism) 54

O

online usage messages 113
onnode (Shell built-in) 148
ontty (Shell built-in) 148
open control blocks 63 – 64
open resources
 active information contained in OCBs 64
OS
 images 83
 partitions 78, 81
output
 appending to a file 113, 137
 redirecting 74, 113, 137 – 138
 resuming 106
 suspending 106
output characters, translation of 86
overwriting
 characters 88, 102

P

parallel devices 96
parallel ports
 determining if port is available 96
 installing 96
parameter expansion (Shell) 130
parameters (Shell)
 list of parameters 142
 named 127, 131
 positional 130
 special 130
parent directory 72

parity error, signal generated 27
partitions 78
 bitmap 83
 creating, modifying, or deleting 79
 flagging as bootable 79
 key components 82 – 83
 loader block 82
 root block 83
 root directory, special characteristics 83
 types 78
PATH (environment variable) 128
pathname prefix tree 58
pathname space
 location 58
 regions of authority 58
pathnames 69, 119
 converting relative pathnames to
 network pathnames 60
 network pathnames 119
 resolving 58 – 59
pattern-matching characters 114, 129
pending signals 25
performance
 device subsystem 97
 fast interactive response via
 adaptive scheduling 38
 Filesystem Manager 75 – 76
 realtime 40 – 43
pipe symbol 113
pipelines 142
pipes 68, 74, 134
 creating 74
 effect of writing on a pipe with no readers 26
 see also FIFOs
portability
 filenames 118
 portable character set 118
positional parameters (Shell) 130
 setting/unsetting 153
 shifting 154

POSIX

- background information viii
- getting information on POSIX viii
- prefix** (utility) 59
- prefix aliases 60 – 62
- prefix tree 58
 - displaying 59
 - inheritance of private 63
 - managed as a list of prefixes 59
 - private 62
 - system tree 62
- prefixes 58
- printing contents of files 122
- priorities 35
 - determining 36
 - setting 36
- process control environments 9
- process creation primitives 46
- process inheritance 47
- Process Manager
 - general responsibilities 46
- process name locator 52 – 53
- process names
 - attaching 53
 - global names 52
 - local names 52
 - locating processes by name 53
 - reducing potential name conflicts 53
- process states
 - DEAD 50
 - determining 52
 - READY 50
 - RECEIVE blocked 15, 50
 - REPLY blocked 15, 50
 - SEND blocked 15, 50
 - SIGNAL blocked 30, 50
 - STOPPED 50
 - WAIT blocked 50

processes

- catching signals 28
- child processes running concurrently
 - with parents 48
- cooperating 13, 52
- cooperating via pipes and FIFOs 74
- creating 48
- creating on any node 47, 62
- creation primitives 46
- death of parent process 48
- determining states 52
- effects of signals 25
- executing 48
- executing on remote nodes 139
- inheritance 47
- life cycle 48 – 49
- loading 48
- locating 52
- locating by name 53
- opening a file 58
- opening the same file twice 64
- priorities, determining 36
- priorities, setting 36
- private prefix tree 62
- process states 50 – 51
- protection against signals 27
- remote processes, passing names to 63
- returning exit status 150
- scheduling 35 – 39
 - scheduling (adaptive) 38
 - scheduling (client-driven) 39
 - scheduling (FIFO) 37
 - scheduling (round-robin) 38
- signal generated upon death of
 - controlling process 26
- signal generated when child process
 - terminates 26
- signaling 147
- signals, receiving 25
- starting on another tty 148

- starting on remote nodes 148
- states 50 – 51
- symbolic names 52
- synchronizing via message passing 15
- system processes 6
- terminating via **kill** built-in 147
- termination 48
- termination by a signal 49
- termination, effect on virtual circuits 31
- termination, effects of 48
- virtual processes 31
- zombie processes 49 – 50
- prompts (Shell) 136
- proxies
 - creating 23
 - processes that can use a proxy 23
 - purpose of 13, 23
 - used by interrupt handlers 23
- PS1 (environment variable) 128
- PS2 (environment variable) 128

Q

- QNX
 - defined 4
 - extensibility 6
 - flexibility 4
 - microkernel architecture 5 – 6
 - network as homogeneous set
 - of resources 8 – 10
 - network flexibility 9
 - network transparency 9
 - single-computer model 9
 - suitability for realtime applications 4
- QNX console 92 – 94
- qnx** terminal 157
 - composed character input sequences 164
 - escape sequences 165

- keyboard codes 162
- standard console character set 160
- quoting (Shell) 135 – 136

R

- ramdisk (Filesystem Manager) 76
- raw device, reading from 89
- raw input mode 89
- raw volumes 78 – 81
 - see also* disks
 - see also* partitions
- read** (Shell built-in) 149
- read operations (**Dev**)
 - non-blocking reads 87
 - non-blocking writes 87
 - parameters defining when read request
 - to be satisfied 89
 - reading console screen 94
 - reading from a raw device 89
 - timed read operations 87
- read operations (**fsys**)
 - priority over write operations 76
 - synchronous 75
- reading
 - input from a file 113
 - output to a file 113
- reading directory entries 70
- readonly** (Shell built-in) 152
- READY (process state) 50
- realtime performance 4, 40 – 43
 - interrupt latency 40
 - scheduling latency 41
 - stacked interrupts 42
- rebooting 106
 - signal generated 27
- recalling commands 103
- RECEIVE blocked (process state) 15, 50

Receive() (message-passing primitive) 13
 using 17
 recovering your filesystem 77
 redirecting
 input 74, 113, 137
 output 74, 113, 137
 redirection operators (Shell) 143
 regular files 68 – 69
 extents 72
 relative pathnames
 converting to network pathnames 60
 remote processes
 passing a name to 63
 removing
 directories 122
 files 122
 timers 55
 renaming files 122
 REPLY blocked (process state) 15, 50
Reply() (message-passing primitive) 13, 18
 using 17
 reply-driven messaging 18
 reserved words (Shell) 140
 resolving pathnames 58 – 59
 resources
 accessing on other machines 8
 open 64
 resuming output 106
return (Shell built-in) 152
 right arrow key 102
 ringing a bell on a terminal 167
 root
 network root 59
 root block 83
 root directory
 special characteristics 83
 round-robin scheduling (processes) 38
 Rubout key 102

S

scheduling latency 41
 scheduling processes 35 – 39
 adaptive scheduling 38
 client-driven priority scheduling 39
 FIFO scheduling 37
 process priorities 35
 round-robin scheduling 38
 scheduling method, determining 37
 scheduling method, setting 37
 scheduling methods 36
 screen size, changing 94, 105
 seek points 64 – 65
 SEND blocked (process state) 15, 50
Send() (message-passing primitive) 13
 using 16
 serial channels
 managing 95
 serial devices 95
 serial ports
 installing 95
 seeing which ports are available 95
 services, handled by QNX processes 6
set (Shell built-in) 153
 Shell
 . (dot) (built-in) 151
 : (colon) (built-in) 150
 arithmetic expansion 128
 break (built-in) 150
 built-in commands (regular) 146
 built-in commands (special) 150
 capabilities 125
 case conditional construct 144
 cd (built-in) 146
 command (built-in) 146
 command basics 109
 command lists 126
 command substitution 127
 commands 143 – 155

- commands, executing 151
- commands, structure 110
- continue** (built-in) 151
- displaying command usage 113
- environment variables 127
- environment, modifying 127
- eval** (built-in) 151
- exec** (built-in) 151
- exit** (built-in) 152
- export** (built-in) 152
- false** (built-in) 146
- file descriptors 138 – 139, 151
- filename expansion 129
- for** loop 145
- function definition command 145
- functions, returning from 152
- functions, unsetting 154
- getopts** (built-in) 147
- here-document 137
- if** (conditional construct) 144
- input, redirecting 137
- kill** (built-in) 147
- loops, continuing 151
- loops, exiting 150
- modifying shell environment 127
- multiple commands, entering 126
- multiple files, specifying 129
- named parameters 127, 131
- named parameters, unsetting 154
- onnode** (built-in) 148
- ontty** (built-in) 148
- options, setting/unsetting 153
- output, redirecting 137
- parameter expansion 130
- pattern-matching characters 129
- pipes 134
- positional parameters 130, 153 – 154
- prompts 136
- quoting 135 – 136
- read** (built-in) 149
- readonly** (built-in) 152
- redirecting input and output 137 – 138
- remote execution of programs 139
- reserved words 140
- return** (built-in) 152
- scripts 140
- set** (built-in) 153
- shift** (built-in) 154
- signals 141
- signals, trapping 154
- special parameters 130
- substituting a command with its **stdout** 127
- terms defined 142
- tilde expansion 132
- trap** (built-in) 154
- true** (built-in) 149
- umask** (built-in) 149
- unset** (built-in) 154
- until** loop 145
- wait** (built-in) 150
- while** loop 145
- wildcards 129
- SHELL (environment variable) 128
- shift** (Shell built-in) 154
- shutdown** (utility)
 - signal generated 27
- SIGNAL blocked (process state) 50
- signal handlers 25, 28 – 29
- signal handling functions 28
- signals
 - blocking delivery 29
 - catching 28
 - default actions 25
 - defining signal processing 28
 - delivery of 24 – 25
 - effect when process is SIGNAL blocked 31
 - effects of ignoring signals 25
 - effects on processes 25
 - effects when process is REPLY blocked 30
 - functions safe to use within signal handlers 28

- generating 24
- messages and signals 30
- order of delivery 25
- pending 25
- processes protecting themselves
 - against signals 27
- processes setting signals on themselves 24
- purpose of 13, 24
- receiving 25
- sending 147
- summary of signals 26
- trapping 141, 154
- single-computer model 9
- slay** (utility) 24
- sleep until completion
 - (event-reporting mechanism) 54
- software interrupts
 - see* signals
- sorting files 122
- spawn()* (process creation primitive) 46, 62
- special parameters (Shell) 130
- stacked interrupts 42
- standard input
 - reading from 149
 - redirecting 113, 137 – 138
- standard output
 - appending to a file 113, 137
 - redirecting 113, 137 – 138
- states of processes
 - see* process states
- STOPPED (process state) 50
- string patterns, searching for 122
- suffixes (filenames) 118
- suspending output 106
- symbolic names
 - see* process names
- synchronizing processes 15
- system consoles 92 – 94
- system managers 6
 - similarity to user-written processes 6

- see also* Device Manager; Filesystem Manager;
Network Manager; Process Manager
- system tree 62

T

- temporary files (Filesystem Manager) 76
- TERM (environment variable) 128
- terminal (**qnx** terminal) 157
- terminal control structure (*termios*) 86
- terminal devices
 - accessing 86
 - device services 86
 - located in I/O namespace 86
 - names 86
 - representing bidirectional stream of
 - bytes to processes 86
 - terminals connected via a serial channel 88
 - see also* devices
- terminal initialization utility (**tinit**) 104
- terminals
 - attached terminals, keyboard limitations 103
 - ringing a bell 167
 - signal generated upon hangup on
 - controlling terminal 26
 - terminal escape sequences 165
- terminating processes 48
 - see also* processes
- termios control structure
 - ICANON bit 87 – 89
- termios parameters
 - functionality controlled by 86
- text, entering **vii**
- tilde expansion (Shell) 132
- time management 53
 - see also* timers
- time stamps 68

timers 53 – 54
 advanced timing 54
 arming 54
 creating 54
 reading 55
 removing 55
 setting resolution 55
 simple timing 53
 timer interrupt handlers 56
tinit (utility) 104
TMPDIR (environment variable) 128
tone, generating on console 167
transparency of QNX networks 9
trap (Shell built-in) 154
true (Shell built-in) 149
true value, returning 149
TZ (environment variable) 53, 128

U

umask (Shell built-in) 149
unset (Shell built-in) 154
until loop (Shell) 145
up arrow key 103
usage messages 113
use command 113
utilities
 cat 122
 cd 122
 chkfsys 77 – 78
 common file management utilities 122
 cp 122
 diff 122
 dinit 82
 displaying usage messages 113
 fdisk 79
 find 122
 grep 122
 kbd 97

kedit 97
 kill 24
 less 122
 ln 71
 lp 122
 ls 93, 95 – 96, 122
 mkdir 122
 mkfifo 74
 more 122
 mount 80
 mv 122
 nameloc 53
 netpoll 34
 parsing utility options 147
 prefix 59
 ps 52
 pwd 122
 qtalk 88
 rm 72, 74, 122
 rmdir 122
 shutdown 27
 sin 52
 slay 24
 sort 122
 stty 86, 88
 ticksize 56
 tinit 104
 wc 122
 who 122
 see also commands; Shell built-in utilities

V

virtual circuits 31
 automatic resizing of buffers 31
 automatic resizing to accommodate
 large messages 31
 created on behalf of the application 33
 effect of terminating process on 31

- integrity check 33
- setting up 31
- sharing 31
- terminating 33
- virtual consoles
 - changing size of current console 94
 - defined 92
 - determining number available 93
 - specifying number of 93
 - switching 93, 104
- virtual processes 31

W

- wait** (Shell built-in) 150
- WAIT blocked (process state) 50
- while** loop (Shell) 145
- wildcards 114, 129
- window change, signal generated 27
- write operations (**Dev**)
 - writing to console screen 94
- write operations (**Fsys**)
 - asynchronous 75
 - 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 19
- zero-length replies 19
- zombie processes 50



