QNX SOFTWARE SYSTEMS

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/intro.html

The Philosophy of QNX

This chapter covers the following topics:

- What is QNX?
- QNX's microkernel architecture
- Interprocess communication (IPC)
- QNX as a network

What is QNX?

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

Some environments call for more rigorous resource management and scheduling than others. Realtime applications, for instance, depend on the operating system to handle multiple events within fixed time constraints. The more responsive the OS, the more "room" a realtime application has to maneuver when meeting its deadlines.

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

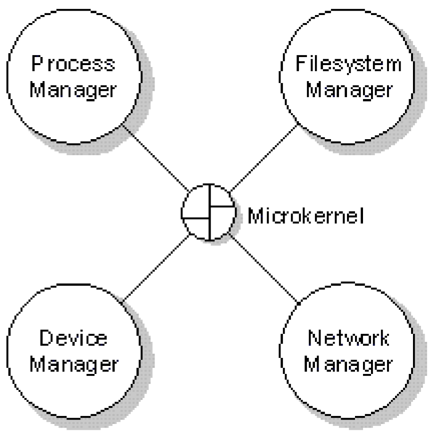
QNX is also remarkably flexible. Developers can easily customize the operating system to meet the needs of their application. From a "bare-bones" configuration of a kernel with a few small modules to a full-blown network-wide system equipped to serve hundreds of users, QNX lets you set up your system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

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

- microkernel architecture
- message-based interprocess communication

QNX's microkernel architecture

QNX consists of a small kernel in charge of a group of cooperating processes. As the following illustration shows, the structure looks more like a team than a hierarchy, as several players of equal rank interact with each other and with their "quarterback" kernel.



The QNX Microkernel coordinating the system managers.

A true kernel

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

But the QNX Microkernel is truly a kernel. First of all, like the kernel of a realtime executive, the QNX Microkernel is very small. Secondly, it's dedicated to only two essential functions:

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

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

System processes

All QNX services, except those provided by the Microkernel, 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 (IPC)

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.

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.

QNX as a message-passing operating system

QNX was the first commercial operating system of its kind to make use of message passing as the fundamental means of IPC. 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 Microkernel can schedule all processes as efficiently as possible to make the most of available CPU resources. This single, consistent method - message-passing - is thus constantly operative throughout the entire system.

Realtime and other mission-critical applications generally require a dependable form of IPC, because the processes that make up such applications are so strongly interrelated. The discipline imposed by QNX'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 - 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.

Single-computer model

QNX is designed from the ground up as a network-wide operating system. In some ways, a QNX network feels more like a mainframe computer than a set of micros. Users are simply aware of a large set of resources available for use by any application. But unlike a mainframe, QNX provides a highly responsive environment, since the appropriate amount of computing power can be made available at each node to meet the needs of each user.

In a process control environment, for example, PLCs and other realtime I/O devices may require more resources than other, less critical, applications, such as a word processor. The QNX network is responsive enough to support both types of applications *at the same time* - QNX lets you focus computing power on the plant floor where and when it's needed, without sacrificing concurrent connectivity to the desktop.

Flexible networking

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

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

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

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

QNX, on the other hand, is grounded on the principle that effective communication is the key to effective operation. Message passing thus forms the cornerstone of QNX's architecture and enhances the efficiency of *all* transactions among all processes throughout the entire system, whether across a PC backplane or across a mile of coax.

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

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/microkernel.html

The Microkernel

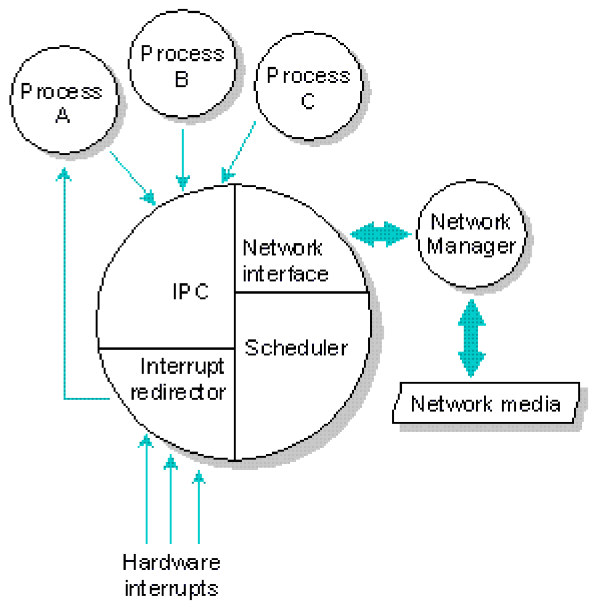
This chapter covers the following topics:

- Introduction
- Interprocess communication
- IPC via messages
- IPC via proxies
- IPC via signals
- IPC across the network
- IPC via semaphores
- Process scheduling
- A word about realtime performance

Introduction

The QNX Microkernel is responsible for the following:

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



Inside the QNX Microkernel.

Interprocess communication

The QNX Microkernel supports three essential types of IPC: messages, proxies, and signals.

- *Messages* - 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* - 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* - a traditional form of IPC. They're used to support asynchronous interprocess communication.

IPC via messages

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

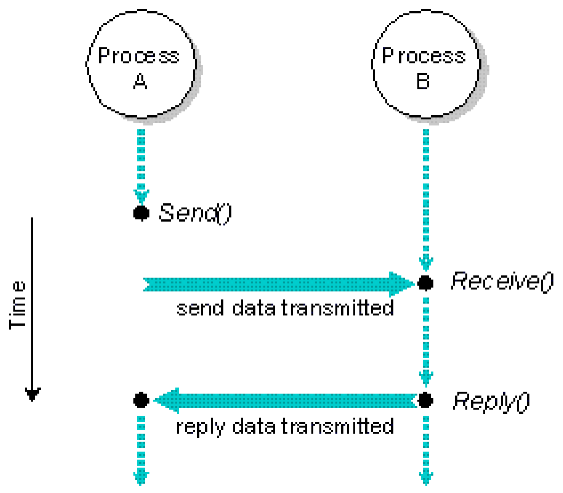
Message-passing primitives

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

C Function:	Purpose:
Send()	to send messages
Receive()	to receive messages
Reply()	to reply to processes that have sent messages

These functions may be used locally or across the network.

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



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

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

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

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

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

Process synchronization

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

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

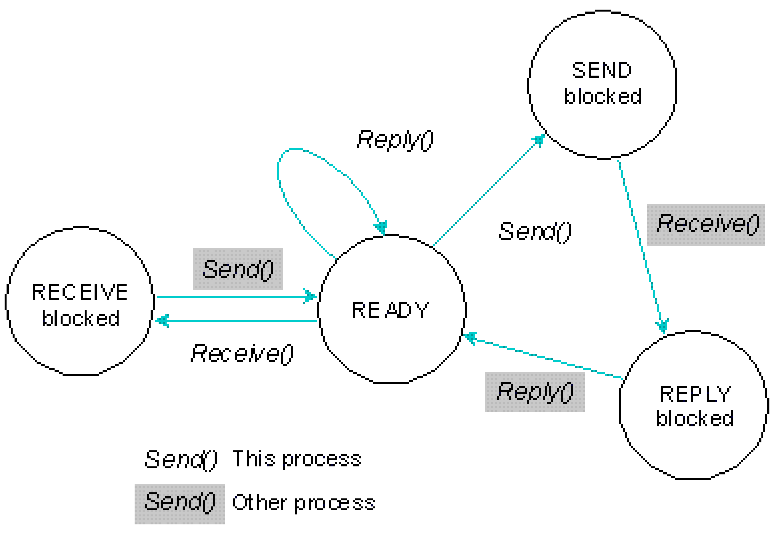
For details on how QNX schedules processes, see "Process scheduling" in this chapter.

Blocked states

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

The following table summarizes the blocked states of processes:

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



A process undergoing state changes 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()*

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.

Send()

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 differ in size, the smaller of the two determines the amount of data that will be transferred.

Reply()

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

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

The *Reply()* call contains these arguments:

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 differ in size, the smaller of the two determines how much data will be transferred.

Reply-driven messaging

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

Although not as common as send-driven messaging, another form of messaging is also possible - and often desirable - to use: *reply-driven messaging*, in which the action is initiated with a *Reply()* instead. Under this method, a "worker" process sends a message to the server indicating that it's available for work. The server doesn't reply immediately, but rather "remembers" that the worker has sent an arming message. At some future time, the server may decide to initiate some action by replying to the available worker process. The worker process will do the work, then finish the action by sending a message containing the results to the server.

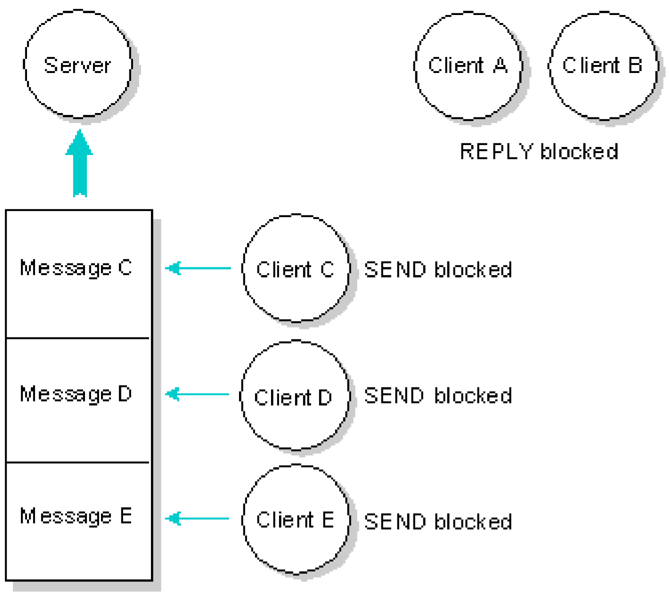
Additional points to remember

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

- The message data is maintained in the sending process until the receiver is ready to process the message. There is *no* copying of the message into the Microkernel. 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. Server has not yet *received messages from Client C, Client D, and Client E.*

Advanced facilities

QNX also provides these advanced message-passing facilities:

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

Conditional message reception

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

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

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

Reading or writing part of a message

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

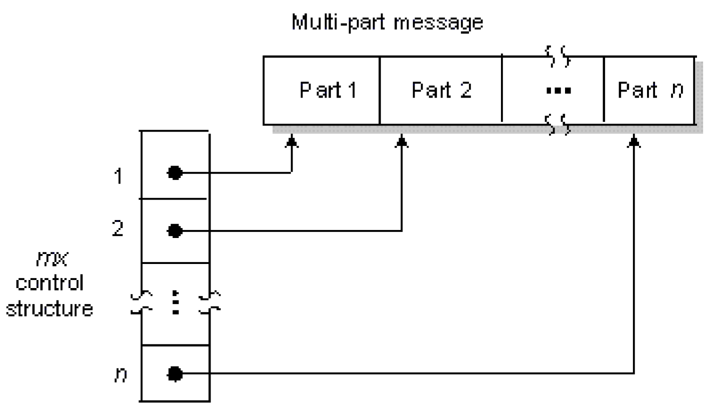
For example, an I/O manager may accept messages of data to be written that consist of a fixed-size header followed by a variable amount of data. The header contains the byte count of the data (0 to 64K bytes). The I/O manager may elect to receive only the header and then use the *Readmsg()* function 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()* 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.

Multipart messages

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

The following functions are available to handle multipart messages:

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



Multipart messages can be specified with an mx control structure. The Microkernel assembles these into a single data stream.

Reserved message codes

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

Reserved range:	Description:
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 QNX system processes

IPC via proxies

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

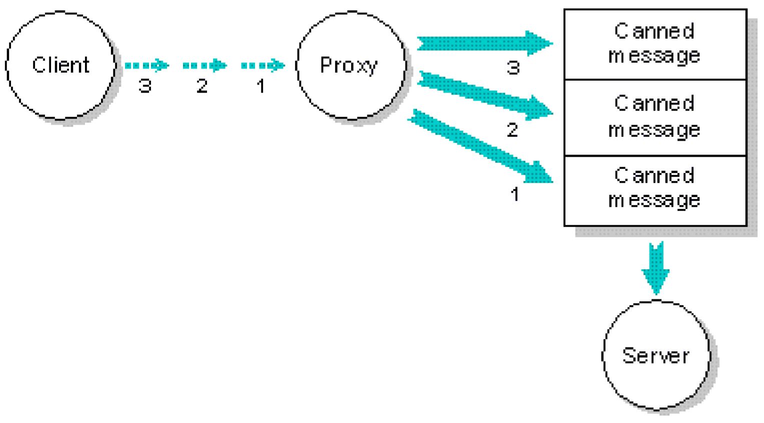
By using a proxy, a process or an interrupt handler can send a message to another process without blocking or having to wait for a reply.

Here are some examples of when proxies are used:

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

Proxies are created with the *qnx_proxy_attach()* 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()* function. The Microkernel 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, some historical UNIX signals, as well as some QNX-specific signals.

Generating signals

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

If you want to:	Use the:
Generate a signal from the shell	`kill` or `slay` utilities
Generate a signal from within a process	*kill()* or *raise()* C functions

Receiving signals

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

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

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

Summary of signals

Signal:	Description:
SIGABRT	Abnormal termination signal such as issued by the *abort()* function.
SIGALRM	Timeout signal such as issued by the *alarm()* function.
SIGBUS	Indicates a memory parity error (QNX-specific interpretation). Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGCHLD	Child process terminated. The default action is to ignore the signal.
SIGCONT	Continue if HELD. The default action is to ignore the signal if the process isn't HELD.
SIGDEV	Generated when a significant and requested event occurs in the Device Manager
SIGFPE	Erroneous arithmetic operation (integer or floating point), such as division by zero or an operation resulting in overflow. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGHUP	Death of session leader, or hangup detected on controlling terminal.
SIGILL	Detection of an invalid hardware instruction. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGINT	Interactive attention signal (**Break**)
SIGKILL	Termination signal - should be used only for emergency situations. *This signal cannot be caught or ignored*. Note that a server with superuser privileges may protect itself from this signal via the *qnx_pflags()* function.
SIGPIPE	Attempt to write on a pipe with no readers.
SIGPWR	Soft boot requested via **Ctrl** -**Alt** -**Shift** -**Del** or `shutdown` utility.
SIGQUIT	Interactive termination signal.
SIGSEGV	Detection of an invalid memory reference. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGSTOP	HOLD process signal. The default action is to hold the process. Note that a server with superuser privileges may protect itself from this signal via the *qnx_pflags()* function.
SIGTERM	Termination signal
SIGTSTP	Not supported by QNX.
SIGTTIN	Not supported by QNX.
SIGTTOU	Not supported by QNX.
SIGUSR1	Reserved as application-defined signal 1
SIGUSR2	Reserved as application-defined signal 2
SIGWINCH	Window size changed

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 is running (including library functions).

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

Blocking signals

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

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

Some UNIX systems have a flawed implementation of signal handlers in that they reset the signal to the default action rather than block the signal. As a result, some UNIX applications call the *signal()* function within the signal handler to re-arm the handler. This has two windows of failure. First, if another signal arrives while your program is in the handler but before *signal()* is called, your program may be killed. Second, if a signal arrives just after the call to *signal()* in the handler, you might enter your handler recursively. QNX supports signal blocking and therefore avoids these problems. You don't need to call *signal()* within your handler. If you leave your handler via a long jump, you should use the *siglongjmp()* function.

Signals and messages

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

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

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

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

- Complete the original request normally - the sender is assured that the message was handled properly.

  OR

- Release any resources tied up and return an error indicating that the process was unblocked by a signal - the sender receives a clear-error indication.

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

IPC across the network

Virtual circuits

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

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

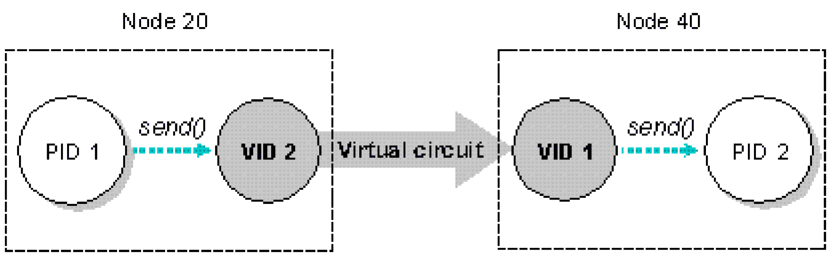
VCs contribute to efficient overall use of resources in a QNX network for several reasons:

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

Virtual processes

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

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



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

Each VID maintains a connection that contains the following information:

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

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

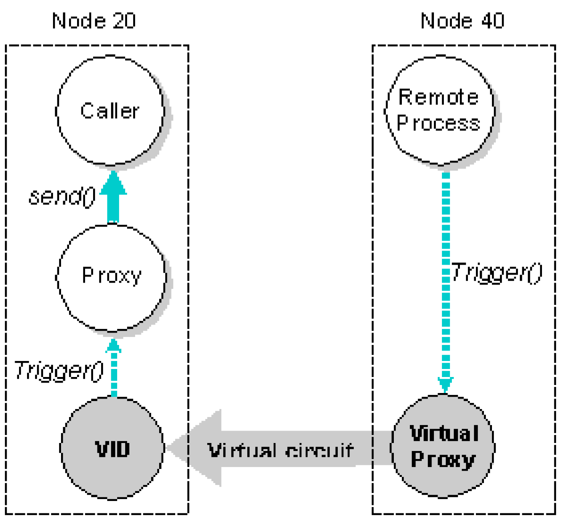
For more information on *qnx_name_locate()*, see the discussion of process symbolic names in Chapter 3.

Virtual proxies

A *virtual proxy* allows a proxy to be triggered from a remote node, much like a virtual circuit allows a process to exchange messages with a remote node.

Unlike a virtual circuit, which binds two processes together, a virtual proxy allows any process on the remote node to trigger it.

Virtual proxies are created by *qnx_proxy_rem_attach()*, which takes a node (*nid_t*) and proxy (*pid_t*) as arguments. A virtual proxy is created on the remote node, which refers to the proxy on the caller's node.



A virtual proxy is created on the remote node, which refers to the proxy on the caller's node.

Note that the virtual circuit is created automatically on the caller's node by *qnx_proxy_rem_attach()*.

Terminating virtual circuits

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

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

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

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

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

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

IPC via semaphores

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

If you wait on a semaphore that's positive, you won't block. Waiting on a non-positive semaphore will block until some other process executes a post. It's valid to post one or more times before a wait - this will allow one or more processes to execute the wait without blocking.

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

Process scheduling

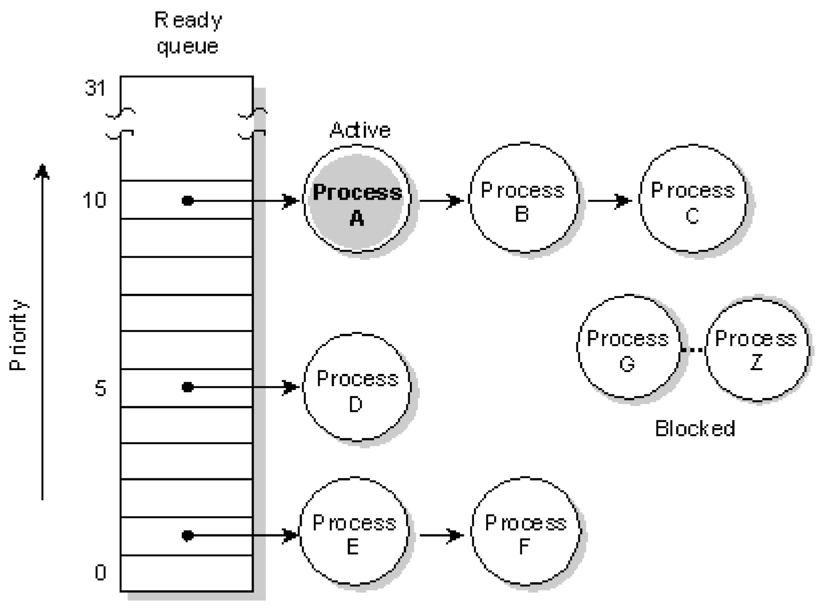
When scheduling decisions are made

The Microkernel's scheduler makes scheduling decisions when:

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

Process priorities

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



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

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

If you want to:	Use this function:
Determine the priority of a process	*getprio()*
Set the priority of a process	*setprio()*

Scheduling methods

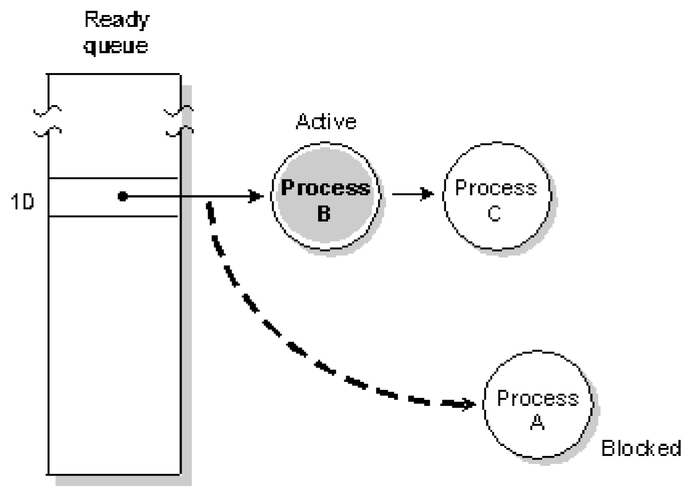
To meet the needs of various applications, QNX provides three scheduling methods:

- 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.

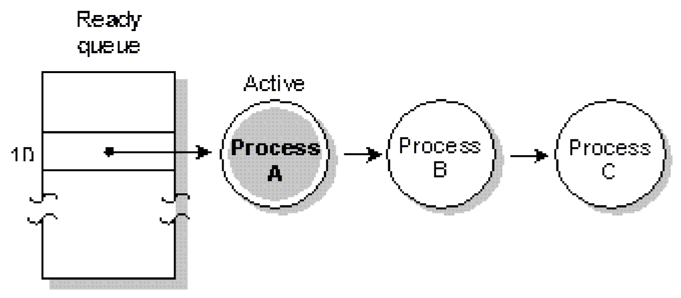
Although a process inherits its scheduling method from its parent process, you can change the method.

If you want to:	Use this function:
Determine the scheduling method for a process	*getscheduler()*
Set the scheduling method for a process	*setscheduler()*

FIFO scheduling

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

- voluntarily relinquishes control (e.g. it makes *any* kernel call)
- is preempted by a higher-priority process



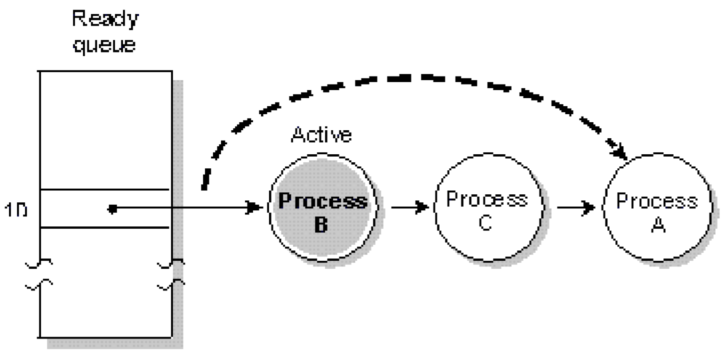
FIFO scheduling. Process A runs until it blocks.

Two processes 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*



Round-robin scheduling. Process A ran until it consumed its timeslice; the next READY process (Process B) now runs.

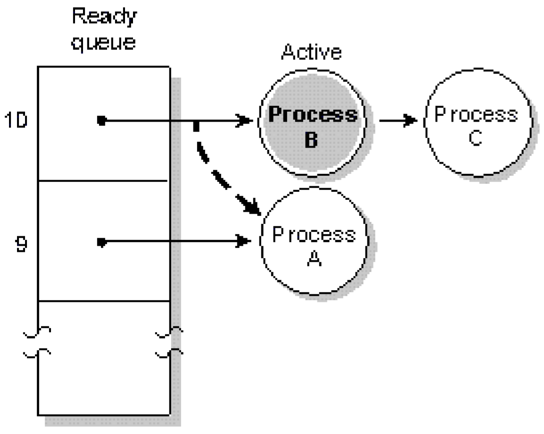
A timeslice is the unit of time assigned to every process. Once it consumes its timeslice, a process is preempted and the next READY process at the same priority level is given control. A timeslice is 50 milliseconds.

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*. Note that a "decayed" process won't continue decaying, even if it consumes yet another timeslice without blocking - it will drop only one level below its original priority.
- If the process blocks, it immediately reverts to its original priority.



Adaptive scheduling. Process A consumed its timeslice; its priority was then dropped by 1. The next

READY process (Process B) runs.

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

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

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, a server will likely run 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 indirectly boosted the priority of the client, because the client's request is what causes the server to run.

As long as the server runs for a short period of time to satisfy the request, this usually isn't a concern. If the server runs for a more extended period, then a low-priority client may adversely affect other processes at priorities higher than the client but lower than the server.

To solve this dilemma, a server may elect to have its priority driven by the priority of the clients that send it messages. When the server receives a message, its priority will be set to that of the client. Note that only its priority is changed - its scheduling method stays the same. If another message arrives while the server is running, the server's priority will be boosted if the new client's priority is greater than the server's. In effect, the new client "turbocharges" the server to its priority, allowing it to finish the current request so it can handle the new client's request. If this weren't done, the new client would 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()* function as follows:

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

A word about realtime performance

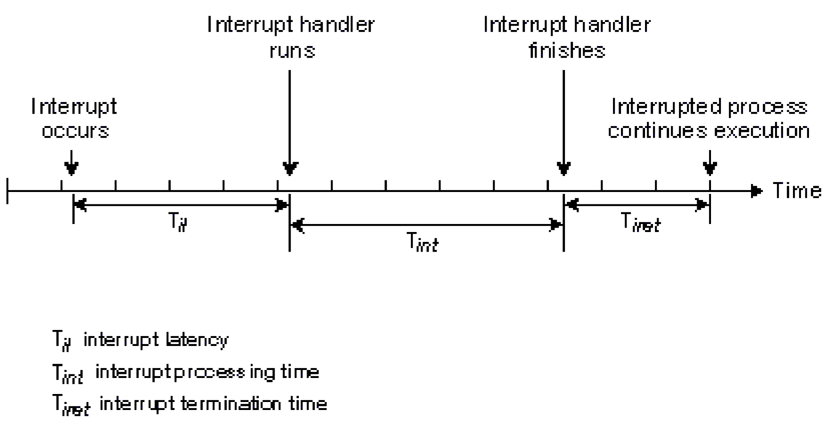
No matter how much we wish it were so, computers are not infinitely fast. In a realtime system, it's absolutely crucial that CPU cycles aren't unnecessarily spent. It's also crucial that you minimize the time it takes from the occurrence of an external event to the actual execution of code within the 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 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 runs
Interrupt handler finishes
Interrupt occurs
Interrupted process continues execution
Time
T_{il}
T_{int}
T_{iret}
T_{il} interrupt latency
T_{int} interrupt processing time
T_{iret} interrupt termination time

Interrupt handler simply terminates.

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

T*il* on various CPUs

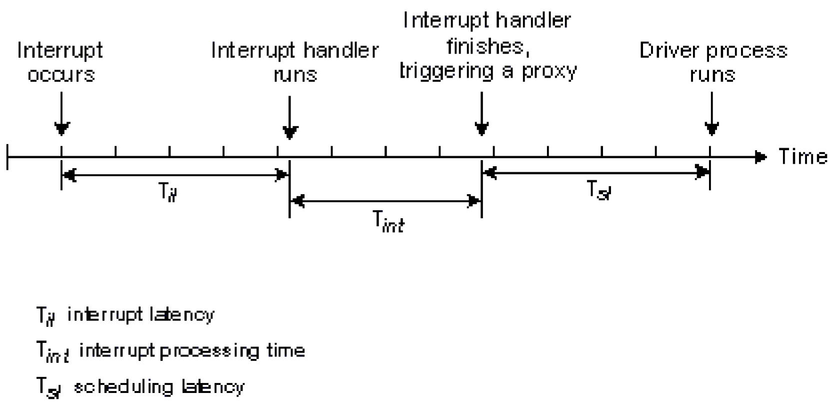
The following table shows typical interrupt-latency times (T*il*) for a range of processors:

Interrupt latency (T*il*):	Processor:
3.3 microsec	166 MHz Pentium
4.4 microsec	100 MHz Pentium
5.6 microsec	100 MHz 486DX4
22.5 microsec	33 MHz 386EX

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

Interrupt handler terminates, triggering a proxy.

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.

T*sl* on various CPUs

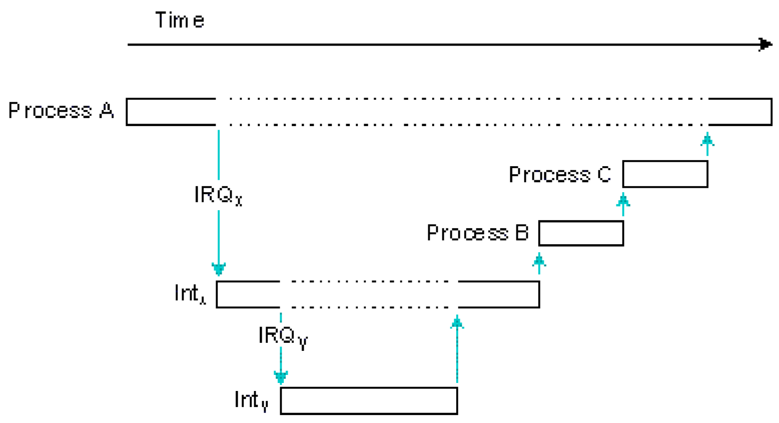
This table shows typical scheduling-latency times (T*sl*) for a range of processors:

Scheduling latency (T*sl*):	Processor:
4.7 microsec	166 MHz Pentium
6.7 microsec	100 MHz Pentium
11.1 microsec	100 MHz 486DX4
74.2 microsec	33 MHz 386EX

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 IRQx causes interrupt handler Intx to run, which is preempted by IRQy and its handler Inty. Inty triggers a proxy causing Process B to run; Intx triggers a proxy causing Process C to run.

QNX Developer Support

The Process Manager

This chapter covers the following topics:

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

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

Process creation primitives

QNX supports three process-creation primitives:

- *fork()*
- *exec()*
- *spawn()*

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

fork()

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

exec()

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

spawn()

The *spawn()* primitive creates a new process as a child of the calling process. It can avoid the need to *fork()* and *exec()*, resulting in a faster and more efficient means for creating new processes. Unlike *fork()* and *exec()*, which by their very nature operate on the same node as the calling process, the *spawn()* primitive can create processes *on any node in the network*.

Process inheritance

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

Item inherited	*fork()*	*exec()*	*spawn()*
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 previous section on "Process inheritance").

Loading

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

Execution

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

Termination

A process is terminated in either of two ways:

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

Termination involves two stages:

1. A *termination thread* in the Process Manager is run. This "trusted" code is in the Process Manager but the thread runs with the process ID of the terminating process. This thread closes all open file descriptors and releases the following:
   - any virtual circuits held by the process
   - all memory allocated to the process
   - any symbolic names
   - any major device numbers (I/O managers only)
   - any interrupt handlers
   - any proxies
   - any timers
2. After the termination thread is run, notification of process termination is sent to the parent process (this phase runs inside the Process Manager).

   If the parent process hasn't issued a *wait()* or *waitpid()* call, the child process becomes a "zombie" process and won't terminate until the parent process issues a *wait()* or terminates. (If you don't want a process to *wait* on the death of children, you should either set the _SPAWN_NOZOMBIE flag with *qnx_spawn()* or set the action for SIGCHLD to be SIG_IGN via *signal()*. This way, children won't become zombies when they die.)

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

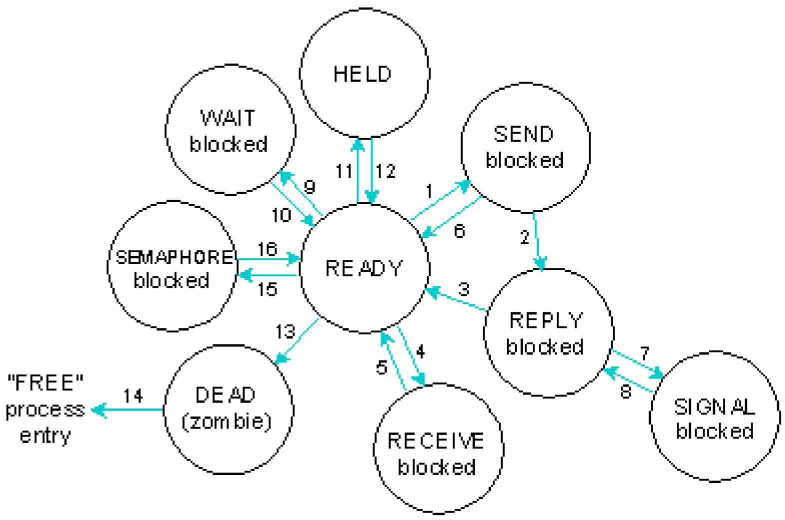
If a process is terminated by the delivery of a termination signal and the `dumper` utility is running, a memory-image dump is generated. You can examine this dump with the symbolic debugger.

Process states

A process is always in one of the following states:

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

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



Possible process states in a QNX system.

The transactions depicted in the previous diagram are as follows:

1. Process sends message.
2. Target process receives message.
3. Target process replies.
4. Process waits for message.
5. Process receives message.
6. Signal unblocks process.
7. Signal attempts to unblock process; target has requested message signal catching.
8. Target process receives signal message.
9. Process waits on death of child.
10. Child dies or signal unblocks process.
11. SIGSTOP set on process.
12. SIGCONT set on process.
13. Process dies.
14. Parent waits on death, terminates itself or has already terminated.
15. Process calls *semwait()* on a non-positive semaphore.
16. Another process calls *sempost()* or an unmasked signal is delivered.

Determining process states

To determine the state of an individual process from within the Shell, use the `ps` and `sin` utilities (from within applications, use the *qnx_psinfo()* function).

To determine the state of the operating system as a whole from within the Shell, use the `sin` utility (from within applications, use the *qnx_osinfo()* function).

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

Process symbolic names

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

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

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

The situation becomes more complex in a network environment where a server may need to service clients from several nodes across a network. QNX accordingly provides the ability to support both global names and local names. Global names are known across the network, whereas local names are known only on the node where they're registered. Global names start with a slash (/). For example:

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

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

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

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

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

Timers

Time management

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

Simple timing facilities

Shell scripts and processes can pause for a specific number of seconds, using a simple timing facility. For Shell scripts, this facility is provided by the `sleep` utility; for processes, it's provided by the *sleep()* function. You can also use the *delay()* 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 1003.1b 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 `Proc` in the *Utilities Reference*). To create timers, you use the *timer_create()* function.

Arming timers

You can arm timers with the following time intervals:

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

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

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

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

To arm timers with:	Use this function:
an absolute or relative time interval	*timer_settime()*

Removing timers

To remove timers, you use the *timer_delete()* function.

Setting timer resolution

You can set the resolution of timers by using the `ticksize` utility or the *qnx_ticksize()* function. You can adjust the resolution from 500 microseconds 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 *timer_gettime()* 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)
- should be as short as possible.

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

If you want to:	Use this function:
Establish a hardware interrupt	*qnx_hint_attach()*
Remove a hardware interrupt	*qnx_hint_detach()*

Timer interrupt handlers

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

You can also attach to a scaled timer interrupt that will activate every 50 milliseconds, regardless of the *tick size*. These timers offer a lower-level alternative to the POSIX 1003.4 timers.

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/namespace.html

I/O Namespace

This chapter covers the following topics:

- Introduction
- Resolving pathnames
- File descriptor namespace

Introduction

I/O namespace

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

Prefixes and regions of authority

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

Resolving pathnames

I/O manager prefixes

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

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

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

The Filesystem Manager has registered two prefixes, one for a mounted QNX filesystem (i.e. `/`) and one for a block special file that represents an entire physical hard drive (i.e. `/dev/hd0`). The character

Device Manager has registered a single prefix. The following table illustrates the longest-match rule for pathname resolution.

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

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

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

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

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

If you want to:	Use the:
Display the prefix tree	`prefix` utility
Obtain the prefix tree from within a C program	*qnx_prefix_query()* function

Network root

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

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

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

Default network root

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

```
//5 ls /
```

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

```
ls /
```

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

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

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

would be resolved as:

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

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

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

```
ls /
```

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

```
ls //9/
```

Likewise, if you were to enter this command:

```
//5 ls /
```

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

If you want to:	Use the:
Get your current default network root	*qnx_prefix_getroot()* function
Set your default network root	*qnx_prefix_setroot()* function
Run a program with a new default network root	on utility

Passing pathnames between processes

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

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

Alias prefixes

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

```
prefix=replacement-string
```

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

```
/=//10/
```

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

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

This new pathname will again be applied against the prefix tree, but this time the prefix tree on node 10 will be used because of the leading `//10` pathname. This will resolve to the Filesystem Manager on node 10, where the *open()* request will be directed. With just a few characters, this alias has allowed us to access a remote filesystem as though it were local.

It's not necessary to run a local filesystem process to perform the redirection. A diskless workstation's prefix tree might look something like this:

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

With this prefix tree, pathnames under `/dev` will be routed to the local character device manager, while requests for other pathnames will be routed to the remote filesystem.

Creating special device names

You can also use aliases to create special device names. For example, if a print spooler were running on node 20, you could alias a local printer pathname to it as follows:

```
/dev/printer=//20/dev/spool
```

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

```
/dev/fd0=//20/dev/fd0
```

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

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

Relative pathnames

Pathnames need not start with a single or a double slash. In such cases, the path is considered *relative* to the current working directory. QNX maintains the current working directory as a character string. Relative pathnames are always converted to full network pathnames by prepending the current working directory string to the relative pathname.

Note that different behaviors result when your current working directory starts with a single slash versus starting with a network root.

Current working directory

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

For example, this command:

```
cd //18/
```

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

On the other hand, this command:

```
cd /
```

would be of the second form, where the default network root would affect the relative pathname resolution. For example, if your default network root were `//9`, then entering `cd dev` would put you in `//9/dev`. Because the current working directory doesn't start with a node override, the default network root is prepended to create a fully specified network pathname.

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

A note about cd

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

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

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

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

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

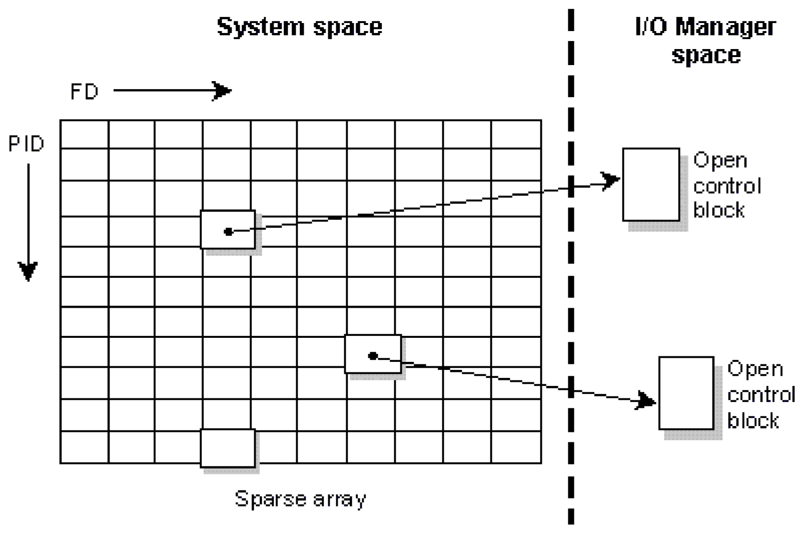
☞ To display a fully resolved network pathname, you can use the `fullpath` utility.

File descriptor namespace

Once an I/O resource has been opened, a different namespace comes into play. The *open()* returns an integer referred to as a *file descriptor* (FD) which is used to direct all further I/O requests to that manager. (Note that the *Sendfd()* kernel call is used within library routines to direct the request.)

The file descriptor namespace, unlike the pathname space, is completely local to each process. The manager uses the combination of a PID and FD to identify the control structure associated with the previous *open()* call. This structure is referred to as an *open control block* (OCB) and is contained within the I/O manager.

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



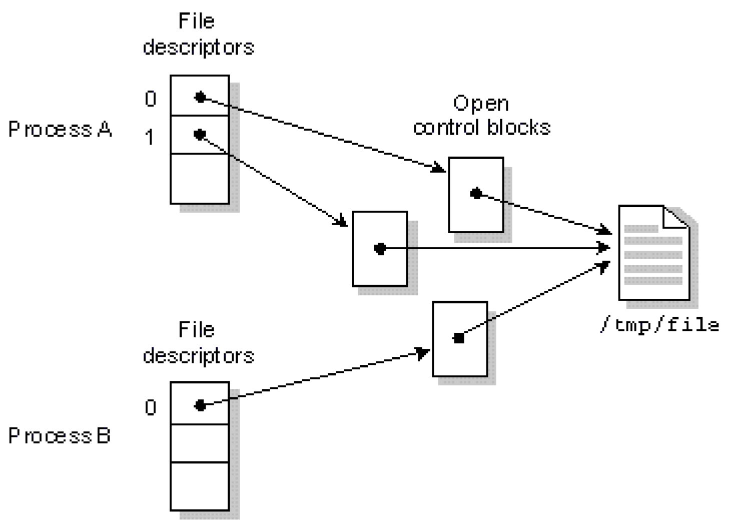
The PID and FD map to an OCB of an I/O Manager.

Open control blocks

The 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 A opens the file /tmp/file twice. Process B 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:

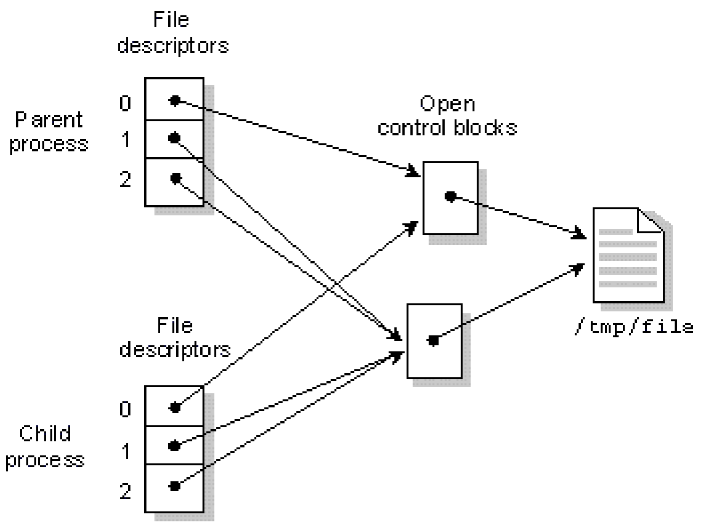
- A process may use the *dup()*, *dup2()*, or *fcntl()* functions to create a duplicate file descriptor that refers to the same OCB.

- When a new process is created via *fork()*, *spawn()*, or *exec()*, all open file descriptors are by default inherited by the new process; these inherited descriptors refer to the same OCBs as the corresponding file descriptors in the parent process.

When several FDs refer to the same OCB, then any change in the state of the OCB is immediately seen by all processes that have file descriptors linked to the same OCB.

For example, if one process uses the *lseek()* function to change the position of the seek point, then reading or writing takes place from the new position no matter which linked file descriptor is used.

The following diagram shows two processes in which one opens a file twice, then does a *dup()* to get a third. The process then creates a child that inherits all open files.



dup() function">

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

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

QNX SOFTWARE SYSTEMS

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/fsys.html

The Filesystem Manager

This chapter covers the following topics:

- Introduction
- What is a file?
- Regular files and directories
- Links and inodes
- Symbolic links
- Pipes and FIFOs
- Filesystem Manager performance
- Filesystem robustness
- Raw volumes
- Key components of a QNX partition
- The DOS Filesystem Manager
- CD-ROM Filesystem
- Flash filesystem
- NFS filesystem
- SMB filesystem

Introduction

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

What is a file?

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

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

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

Date and time stamps

`Fsys` maintains four different times for each file:

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

File access

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

- user only
- group only
- others

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

Regular files and directories

Regular files

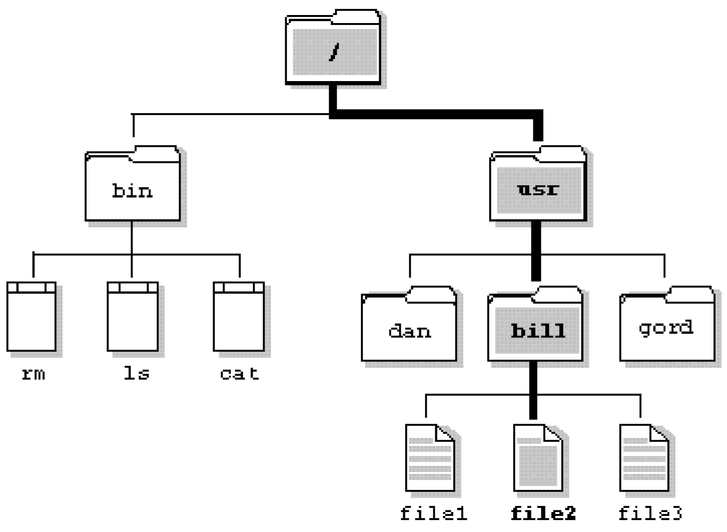
QNX views a *regular file* as a randomly accessible sequence of bytes that has no other predefined internal structure. Application programs are responsible for understanding the structure and content of any specific regular file.

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

Directories

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

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



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

Directory operations

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

Reading directory entries

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

- *opendir()*
- *readdir()*
- *rewinddir()*
- *closedir()*

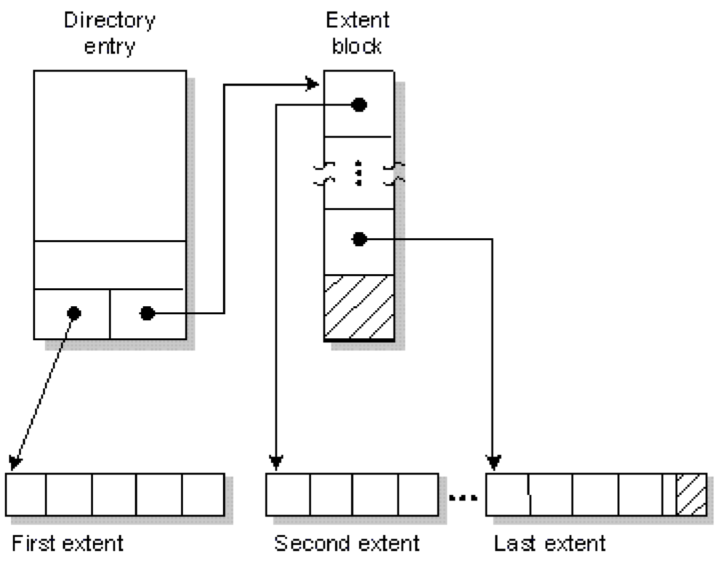
Since QNX directories are simply files that contain "known" information, you can also read directory entries directly using the *open()* and *read()* functions. This technique isn't portable, however-the format of directory entries varies from operating system to operating system.

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.



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

Extending files

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

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

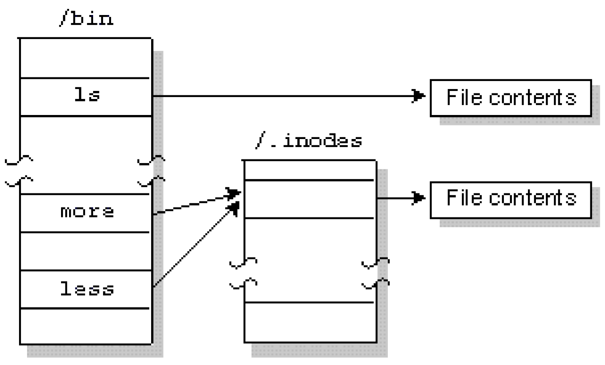
Links and inodes

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

In order to support links for each file, the filename is separated from the other information that describes a file. The non-filename information is kept in a storage table called an *inode* (for "information node").

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

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



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

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

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

If you want to:	Use the:
Create links from within the Shell	`ln` utility
Create links from within programs	*link()* 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 are removed, the link count is decremented. The filespace isn't removed from disk until its link count goes to zero *and* all programs using the file have closed it. This allows an open file to remain in use, even though it has been completely unlinked.

If you want to:	Use the:
Remove links from within the Shell	`rm` utility
Remove links from within programs	*remove()* or *unlink()* functions

Directory links

Although you can't create hard links to directories, each directory has two hard-coded links already built in:

- `.` ("dot")
- `..` ("dot dot")

The filename "dot" specifies the current directory; "dot dot" specifies the directory *above* the current one.

Note that the "dot dot" entry of "/" is simply "/"-you can't go further up the path.

Symbolic links

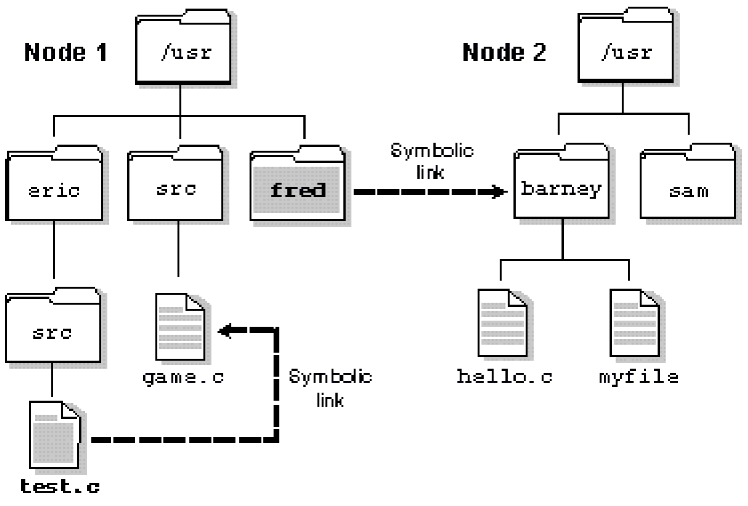
A *symbolic link* is a special file that has a pathname as its data. When the symbolic link is named in an I/O request-by *open()*, for example-the link portion of the pathname is replaced by the link's "data" and the path is re-evaluated. Symbolic links are a flexible means of pathname indirection and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

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

```
//1/usr/fred --> //2/usr/barney
```

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

```
//1/usr/eric/src/test.c --> //1/usr/src/game.c
```



If you want to:	Use this utility:
Create symbolic links	`ln` (with **-s** option)
Remove symbolic links*	`rm`
Query whether a file is a symbolic link	`ls`

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

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

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

Pipes and FIFOs

Pipes

A *pipe* is an unnamed file that serves as an I/O channel between two or more cooperating processes-one process writes into the pipe, the other reads from the pipe. The Filesystem Manager takes care of buffering the data. The buffer size is defined as PIPE_BUF in the `<limits.h>` file. A pipe is removed once both of its ends have closed.

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

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

```
ls | more
```

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

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

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

FIFOs

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

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

Filesystem Manager performance

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

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

Elevator seeking

Elevator seeking minimizes the overall seek time required to read or write data from or to disk. Outstanding I/O requests are ordered such that they can all be performed 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 I/O is performed whenever possible.

Buffer cache

The *buffer cache* is an intelligent buffer between the Filesystem Manager and the disk driver. The buffer cache attempts to store filesystem blocks in order to minimize the number of times the Filesystem Manager has to access the disk. By default, the size of the cache is determined by total system memory, but you can specify a different size via an option to Fsys.

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

Applications can modify write behavior on a file-by-file basis. For example, a database application can cause all writes for a given file to be performed synchronously. This would ensure a high level of file integrity in the face of potential hardware or power problems that might otherwise leave a database in an inconsistent state.

Multi-threading

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

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

Client-driven priority

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

Temporary files

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

Ramdisks

The Filesystem Manager has an integrated ramdisk capability that allows up to 8M of memory to be used as a simulated disk. Since the Filesystem Manager uses highly efficient multipart messaging, data moves from the ramdisk directly to the application buffers.

The Filesystem Manager is able to bypass the buffer cache because the ramdisk is built in, not implemented as a driver. (For information on multipart messaging, see the section on "Advanced facilities" in Chapter 2.)

Because they eliminate the delays of physical hardware and don't rely on the filesystem cache, ramdisks provide greater determinism in read/write operations than hard disks.

Filesystem robustness

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

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

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

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

OS partitions

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

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

Type	Filesystem
1	DOS (12-bit FAT)
4	DOS (16-bit FAT; partitions <32M)

5	DOS Extended Partition
6	DOS 4.0 (16-bit FAT; partitions >=32M)
7	OS/2 HPFS
7	Previous QNX version 2 (pre-1988)
8	QNX 1.x and 2.x ("qny")
9	QNX 1.x and 2.x ("qnz")
11	DOS 32-bit FAT; partitions up to 2047G
12	Same as Type 11, but uses Logical Block Address `Int 13h` extensions
14	Same as Type 6, but uses Logical Block Address `Int 13h` extensions
15	Same as Type 5, but uses Logical Block Address `Int 13h` extensions
77	QNX POSIX partition
78	QNX POSIX partition (secondary)
79	QNX POSIX partition (secondary)
99	UNIX

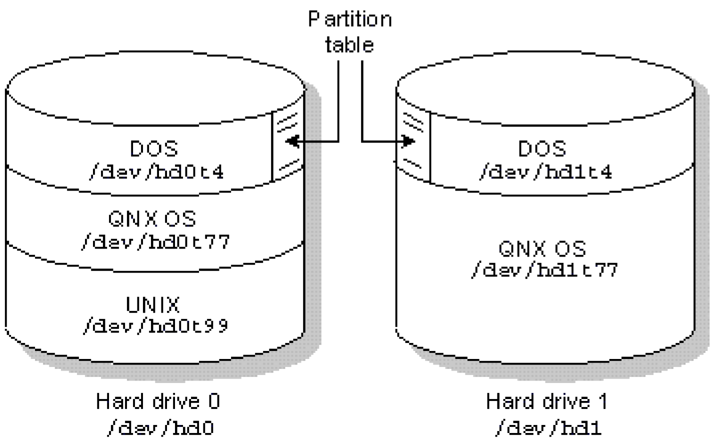
If you want more than one QNX 4.x partition on a single physical disk, you would use type 77 for your first QNX partition, type 78 for your second QNX partition, and type 79 for your third QNX partition. You can use other types for your second and third partitions, but 78 and 79 are preferred. To mark any of these partitions as bootable, you use the `fdisk` utility.

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

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

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

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



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

Defining block special files

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

For example, let's say you have a standard IDE interface 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 a partition block special file (e.g. `/dev/hd0t77` that defines a QNX 4.x partition, not the base block special file that defines the entire raw disk (e.g. `/dev/hd0`. If you attempt to mount the base block special file for the entire disk, you'll get a "corrupt filesystem" message when you try to access the filesystem.

Unmounting a filesystem

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

```
umount /dev/hd0t77
```

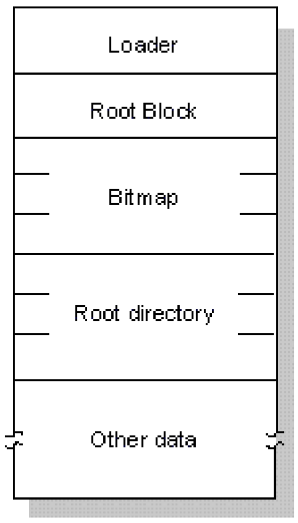
Once a filesystem is unmounted, files on that partition are no longer accessible.

Key components of a QNX partition

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

- loader block
- root block
- bitmap
- root directory

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



Structure of a QNX filesystem within a disk partition.

Loader block

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

Root block

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

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

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

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

Bitmap

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

Root directory

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

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

The DOS Filesystem Manager

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

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

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

Note that `/dos/c` is the pathname of the DOS drive `C`. The `cp` command contains no special code to detect whether the file it's copying is located on a DOS disk. Other commands also work with equal transparency (e.g. `cd`, `ls`, `mkdir`).

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

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

CD-ROM filesystem

The CD-ROM Filesystem, `Iso9660fsys`, provides transparent access to CD-ROM media, so you can treat CD-ROM filesystems as though they were POSIX filesystems. This transparency allows processes to operate on CD-ROM files without any special knowledge or work on their part.

`Iso9660fsys` implements the ISO 9660 standard, including Rock Ridge extensions. DOS and Windows Compact Discs follow this standard. In addition to normal files, `Iso9660fsys` also supports audio.

Flash

The Flash filesystem manager, `Efsys.*`, was explicitly designed to work with Flash memory, whether built-in or removable. Files written to removable Flash media (PC-Card cards) are portable to other systems that also support this standard.

`Efsys.*` combines the functions of a filesystem manager and a device driver to manage a Flash filesystem on memory-based media. Because `Efsys.*` includes the device driver for controlling the hardware, there are separate versions of `Efsys.*` for different embedded systems hardware. For example:

- `Efsys.explr2` for the Intel EXPLR2 evaluation board (with on-board RFA)

- `Efsys.elansc400` for the AMD Élan SC400 evaluation board
- `Efsys.pcmcia` for mixed use of Flash filesystems and PCMCIA devices

Restrictions

The functionality of the filesystem is restricted by the memory devices used. For example, if the medium contains ROM devices only, the filesystem is read-only.

Memory-based Flash devices have restrictions on file writing. You can only append to the file. In addition, file access times aren't updated. Currently, these restrictions apply even if devices such as SRAM are used.

Space reclamation

`Efsys.*` stores directories and files using linked lists of data structures, rather than the fixed-size disk blocks used in traditional rotating-medium filesystems. When a directory or file is deleted, the data structures representing it are marked as deleted, but aren't removed. This avoids continuously erasing and rewriting the medium.

Eventually, the medium will run out of free space and the filesystem manager will perform space reclamation (or "garbage collection"). During this process, `Efsys.*` recovers the space occupied by deleted files and directories. To perform space reclamation, the filesystem manager requires at least one spare block on the medium. The `mkffs` utility automatically reserves one spare block when you make a filesystem.

Compression and decompression

`Efsys.*` supports decompression, which increases the amount of data that can be stored on a medium. Decompression operates transparently to applications.

For files to be decompressed transparently by the filesystem manager, you must compress them explicitly before using `mkffs`. You do this with the `bpe` utility. If you copy a compressed file to a live Flash filesystem, it will remain compressed when it's read from the filesystem.

File permissions

If you disable the POSIX extensions, file ownership is permanently set to `root` and all permissions are set to `rwx`. The `chgrp`, `chmod`, and `chown` commands won't work.

Mounting

You can set mountpoints only when initializing the partitions or when starting the filesystem manager.

Raw device access

When you start `Efsys.*`, it creates a special device file in the `/dev` directory for each medium. In a system with two memory-based media, `Efsys.*` would create the special files `/dev/skt1` and `/dev/skt2`. The special device ignores the partitioning, allowing raw device access to the medium.

An image filesystem partition can be accessed only as a raw device. For each image filesystem partition, `Efsys.*` creates a special file in the format `/dev/skt`*`X`*`img`*`Y`*, where *X* is the medium socket number and *Y* is the number of the image filesystem partition in that medium.

NFS filesystem

Originally developed by Sun Microsystems, the Network File System (NFS) is a TCP/IP application that has since been implemented on most DOS and Unix systems. You should find the QNX implementation invaluable for transparently accessing filesystems on other operating systems that support NFS.

QNX inherently supports networked filesystems. You need to use NFS *only* if you're accessing non-QNX NFS filesystems, or if you want to let NFS clients access your QNX namespace.

NFS lets you graft remote filesystems - or portions of them - onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g. `ls`, `cp`, `mv`) operate on the remote files exactly as they do on your local files.

In QNX 4, NFS requires the `Socket` manager, which implements the networking protocols used in TCP/IP for QNX. Note that a "lighter" version of the socket manager, known as `Socklet`, is also available if you don't need NFS.

SMB filesystem

`SMBfsys` implements the SMB (Server Message Block) file-sharing protocol, which is used by a number of different servers such as Windows NT, Windows 95, Windows for Workgroups, LAN Manager, and Samba. `SMBfsys` allows a QNX client to transparently access remote drives residing on such servers.

`SMBfsys` implements this protocol using NetBIOS on TCP/IP only, *not* NetBEUI. Accordingly, you need TCP/IP installed on both the QNX and remote server side. Once `SMBfsys` is running and a remote server has been mounted, the server's filesystem appears in the local directory structure.

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/dev.html

The Device Manager

This chapter covers the following topics:

- Introduction
- Device services
- Edited input mode
- Raw input mode
- Device drivers
- The QNX console
- Serial devices
- Parallel devices
- Device subsystem performance

Introduction

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

```
/dev/con1
```

Device services

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

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

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

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

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

A process can:	via this C function:
Perform timed read operations	*dev_read()* or *read()* and *tcsetattr()*
Asynchronously notify a process of data available on one or more input devices	*dev_arm()*
Wait for output to be completely transmitted	*tcdrain()*
Send breaks across a communications channel	*tcsendbreak()*
Disconnect a communications channel	*tcdropline()*
Insert input data	*dev_insert_chars()*
Perform non-blocking reads and writes	*open()* and *fcntl()* (O_NONBLOCK mode)

Edited input mode

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

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

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

`Dev` will:	When it receives:
Move the cursor one character to the left	LEFT
Move the cursor one character to the right	RIGHT
Move the cursor to the beginning of the line	HOME
Move the cursor to the end of the line	END
Rub out the character to the left of the cursor	ERASE
Delete the character at the current cursor position	DEL
Rub out the entire input line	KILL
Erase the current line and recall a previous line	UP
Erase the current line and recall the next line	DOWN
Toggle between insert mode and typeover mode (every new line starts in insert mode)	INS

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

If a terminal is connected to QNX via a serial channel, you need to define the editing characters that apply to that particular terminal. To do this, you can use the `stty` utility. For example, if you have a VT100 terminal connected to a serial port (called `/dev/ser1`), you would use the following statement to extract the appropriate editing keys from the *terminfo* database and apply them to `/dev/ser1`:

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

If, instead, you had a modem connected to that serial port, which in turn was connected to another

QNX system running the `qtalk` utility, you'd want to set the line-editing keys as follows:

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

Raw input mode

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

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

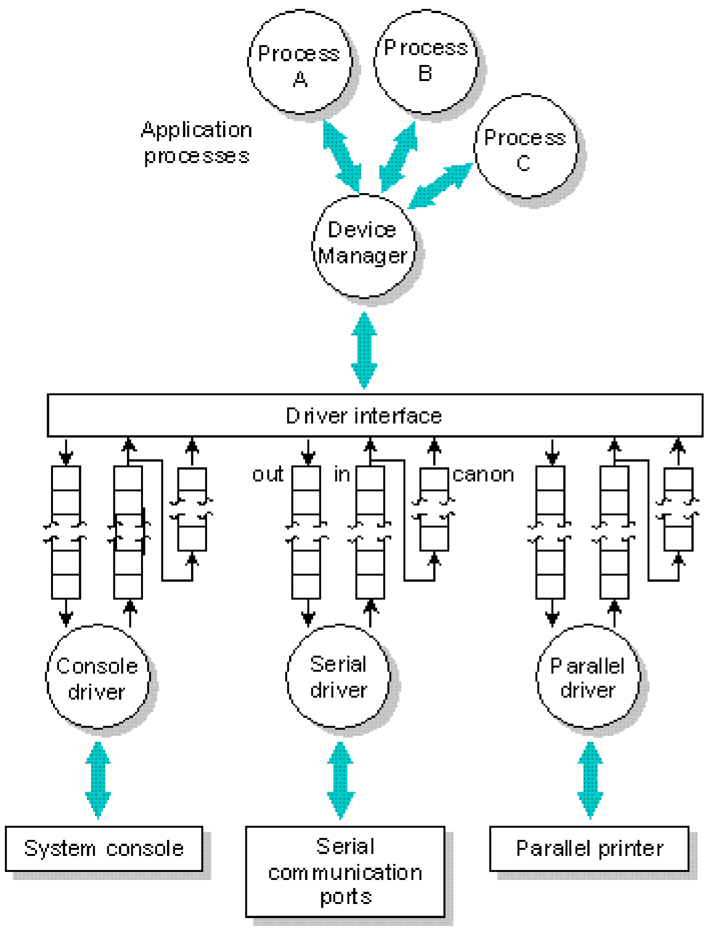
When reading from a *raw* device, the application is capable of specifying under what conditions an input request is to be satisfied. The criteria used for accepting raw input data are based on two members of the *termios* control structure: MIN and TIME. The application can specify a further qualifier of accepting input data when it issues its read request via *dev_read()*. This qualifier, TIMEOUT, is useful when writing protocols or realtime applications. Note that TIMEOUT is always `0` for *read()*.

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

MIN	TIME	TIMEOUT	Description:
`0`	`0`	`0`	Return immediately with as many bytes as are currently available (up to *n* bytes).
`M`	`0`	`0`	Return with up to *n* bytes only, when at least `M` bytes are available.
`0`	`T`	`0`	Return with up to *n* bytes when at least one byte is available, or `T` * .1 second has expired.
`M`	`T`	`0`	Return with up to *n* 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.
`0`	`0`	`t`	RESERVED.
`M`	`0`	`t`	Return with up to *n* bytes when `t` * .1 second has expired, or `M` bytes are available.
`0`	`T`	`t`	RESERVED.
`M`	`T`	`t`	Return with up to *n* bytes when `M` bytes are available, or `t` * .1 second has expired and no characters are received, or at least one byte has been received and the interbyte time between any subsequently received characters exceeds `T` * .1 second.

Device drivers

The following illustration shows a typical QNX device subsystem.



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

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

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.

Device control

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

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

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

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

The QNX console

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

QNX permits multiple sessions to be run concurrently on consoles by means of *virtual consoles*. The `Dev.con` console driver process typically manages more than one set of I/O queues to `Dev`, which are made available to user processes as a set of *terminal devices* with names like `/dev/con1`, `/dev/con2`, etc. From the application's point of view, there "really are" multiple consoles available to be used.

Of course, there's only one *physical* screen and keyboard, so only *one* of these virtual consoles is actually displayed at any one time. The keyboard is "attached" to whichever virtual console is currently visible.

Console-specific functions

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

A process can:	via this C function:
Read directly from the console screen	*console_read()*
Write directly to the console screen	*console_write()*
Be asynchronously notified of significant events (e.g. display data or size has changed, cursor has moved, visible console has changed, etc.)	*console_arm()*
Control the console size	*console_size()*
Switch the 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 provides terminal devices with names such as /dev/ser1, /dev/ser2, etc.

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

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

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

Parallel devices

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

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

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

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

Device subsystem performance

The flow of events within the device subsystem is engineered to minimize overhead and maximize throughput when a device is in *raw* mode. To accomplish this, the following rules are used:

- Interrupt handlers place received data directly into a memory queue. Only when a read operation is pending, *and* that read operation can now be satisfied, will the interrupt handler schedule Dev to run. In all other cases, the interrupt simply returns. Moreover, if Dev is already running, no scheduling takes place.
- When a read operation is satisfied, Dev replies to the application process *directly* from the raw input buffer into the application's receive buffer. The net result is that the data is copied only once.

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

- - -

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/net.html

The Network Manager

This chapter covers the following topics:

- Introduction
- Network Manager responsibilities
- Microkernel/Network Manager interface
- Network drivers
- Node and network identifiers
- Choosing a network
- TCP/IP networking

Introduction

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

- increased throughput via load balancing
- fault tolerance via redundant connectivity
- bridging between QNX networks

Network Manager responsibilities

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

An independent module

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

When the Network Manager starts, it registers with the Process Manager and Microkernel. This activates existing code within the two that interfaces to the Network Manager. This means that network messaging and remote process creation are not just a layer added onto the operating system. Network messaging is integrated into the very heart of the messaging and process-management primitives themselves.

This deep integration at the lowest level gives QNX its network transparency and qualifies it as a fully distributed operating system. Since applications access all services via messaging, and since the Network Manager allows messages to flow transparently on the network, QNX nodes perform together as a single, logical computer.

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

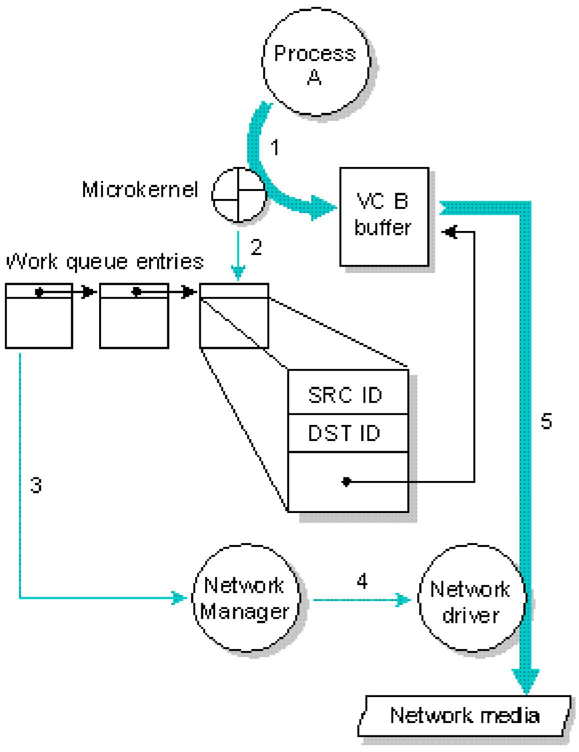
Microkernel/Network Manager interface

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

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

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

Sending a message to a remote node



Send() or *Reply()* to a remote node">

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

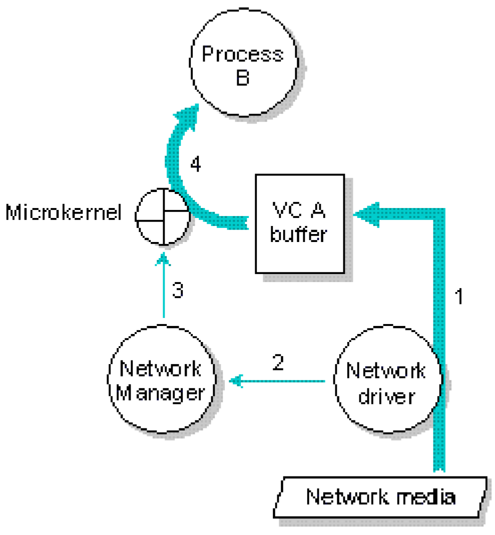
In the case of a *Send()* or *Reply()* to a remote node, the following events occur:

1. The process calls *Send()* or *Reply()* and the Microkernel puts the data from the process's data space into the virtual circuit buffer associated with the VC specified by the *Send()* or *Reply()*.
2. The Microkernel enqueues to the Network Manager a time-ordered queue entry identifying the sender, the remote receiver, and pointers to the data in the virtual circuit buffer. If the queue

was previously empty, the Network Manager's proxy is triggered to let it know new work has arrived.
3. The Network Manager dequeues the queue entry.
4. The Network Manager passes the entry to an appropriate network driver.
5. The network driver begins transmission on the network media and is responsible for delivery.

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

Receiving the message on the remote node



Send() or *Reply()*">

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

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

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

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

Network drivers

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

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

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

Node and network identifiers

Each node in a local area network is identified by two numbers:

- its physical node ID (or IDs if the node has more than one network card)
- a combination of its logical node ID and logical network ID.

Physical node ID

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

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

Logical node ID

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

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

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

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

Note that the logical node ID must be unique for all nodes across *all* interconnected QNX networks in order for network bridging to work.

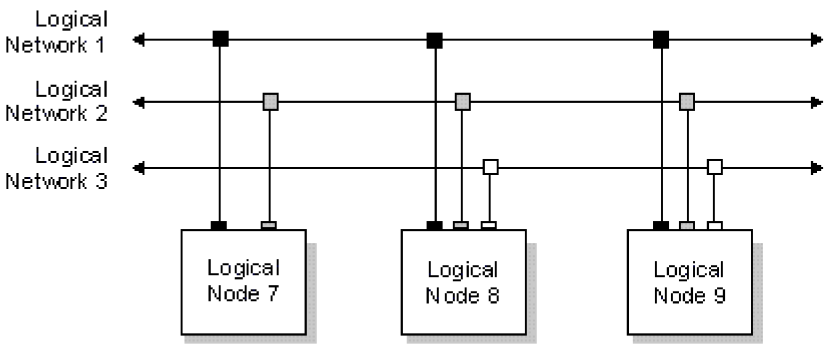
Logical network ID

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

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

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

Logical network and logical node IDs are assigned by the system administrator. For more information, see Network Installation in the *Installation & Configuration* guide.



Multiple physical networks happily coexist via logical networks.

Choosing a network

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

Load balancing

Network throughput is determined by a combination of the speed of the computer and the speed of the network. If your computer can provide data faster than the network can accept it, then the network will limit your throughput.

For example, two Pentium computers connected by a 10BASE-T Ethernet network will be limited to a maximum of 1.1 million bytes per second, which is the data rate provided by the Ethernet hardware. If, however, you were to place two Ethernet cards in each computer and connect them with separate cables, the Network Manager could transmit data over both networks simultaneously. Under a heavy load, this would provide up to twice as much throughput as a single network.

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

Fault tolerance

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

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

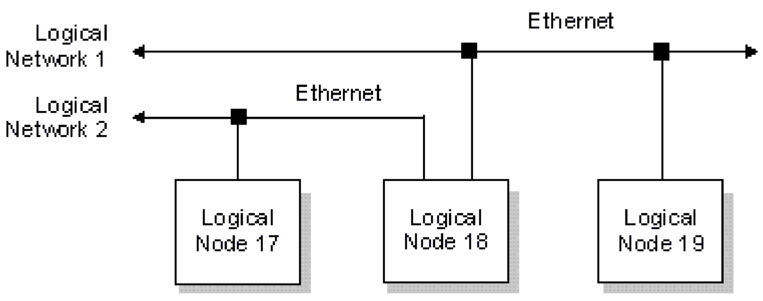
Bridging between QNX LANs

The Network Manager allows any node to act as a bridge between two separate IEEE 802-based QNX networks.

Since QNX uses the same packet format and protocol on all IEEE 802-based networks, you can create bridges between Ethernet, Token Ring, and FDDI networks.

Arcnet networks *cannot* be bridged.

Consider the following diagram, where node 17 and node 18 are on one network, and node 18 and node 19 are on another network:



Network bridging between two IEEE 802-based QNX LANs.

Nodes 17 and 18 are on the same network, so they can talk directly to each other. The same is true for nodes 18 and 19. But how can node 17 talk to node 19?

Since *both* the LANS involved are IEEE 802-based, node 18 automatically bridges packets, allowing node 17 and node 19 to create a virtual circuit. Although they're not connected to the same LAN, nodes 17 and 19 can nevertheless talk to each other as if they were.

TCP/IP networking

QNX's inherent network support implements a LAN that relies on its own *proprietary* protocol and is optimized to provide a fast, seamless interface between QNX computers. But to communicate with non-QNX systems, QNX uses the *industry-standard* set of networking protocols collectively known as TCP/IP.

As the Internet has grown to become more and more visible in our daily lives, the protocol it's based on - IP (Internet Protocol) - has become increasingly important. Even if you're not connecting to "*The* Internet" per se, the IP protocol and tools that go with it are ubiquitous, making IP the de facto choice for many private networks.

IP is used for everything from simple tasks (e.g. remote login) to more complicated tasks (e.g. delivering realtime stock quotes). More and more businesses are turning to the World Wide Web, which commonly rides on IP, for communication with their customers, advertising, and other business connectivity.

TCP/IP Manager

The QNX TCP/IP Manager is derived from the Berkeley BSD 4.3 stack, which is the most common

TCP/IP stack on the Internet and has been used as the basis for many systems.

Socket API

The BSD Socket API was the obvious choice for QNX 4. The Socket API is the standard API for TCP/IP programming in the Unix world. In the Windows world, the Winsock API is based on and shares a lot with the BSD Socket API. This makes conversion between the two fairly easy.

All the routines that application programmers would expect are available, including:

accept()
bind()
bindresvport()
connect()
dn_comp()
dn_expand()
endprotoent()
endservent()
gethostbyaddr()
gethostbyname()
getpeername()
getprotobyname()
getprotobynumber()
getprotoent()
getservbyname()
getservent()
getsockname()
getsockopt()
herror()
hstrerror()
htonl()
htons()
h_errlist()
h_errno()
h_nerr()
inet_addr()
inet_aton()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()
ioctl()
listen()
ntohl()
ntohs()
recv()
recvfrom()
res_init()
res_mkquery()
res_query()
res_querydomain()
res_search()
res_send()
select()
send()
sendto()
setprotoent()
setservent()
setsockopt()
shutdown()

socket()

The common daemons and utilities from the Internet will easily port or just compile in this environment. This makes leveraging what already exists for your applications a snap.

Network interoperability

The QNX TCP/IP Manager was designed and implemented with interoperability utmost in mind. The code takes into account both the RFCs and the real world. The TCP/IP Manager addresses all the functionality proposed in RFC 1122. The ARP, IP, ICMP, UDP, and TCP protocols are supported as well.

NFS

The Network FileSystem (NFS) is a TCP/IP application that has been implemented on most DOS and Unix systems. NFS lets you graft remote filesystems - or portions of them - onto your local namespace. Files on remote systems appear as part of your local QNX filesystem.

In QNX 4, NFS requires the `Socket` manager. Note that a "lighter" version of the socket manager, known as `Socklet`, is also available if you don't need NFS.

SMB

QNX also supports the Server Message Block (SMB) file-sharing protocol, which is used by a number of different servers such as Windows NT, Windows 95, Windows for Workgroups, LAN Manager, and Samba. The `SMBfsys` filesystem allows a QNX client to transparently access remote drives residing on such servers.

QNX SOFTWARE SYSTEMS

QNX Developer Support

http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/photon.html

The Photon microGUI Windowing System

This chapter covers the following topics:

- A graphical microkernel
- The Photon event space
- Graphics drivers
- Scalable fonts
- Unicode multilingual support
- Animation support
- Printing support
- The Photon Window Manager
- Widget library
- Summary

A graphical microkernel

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

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

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

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

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

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

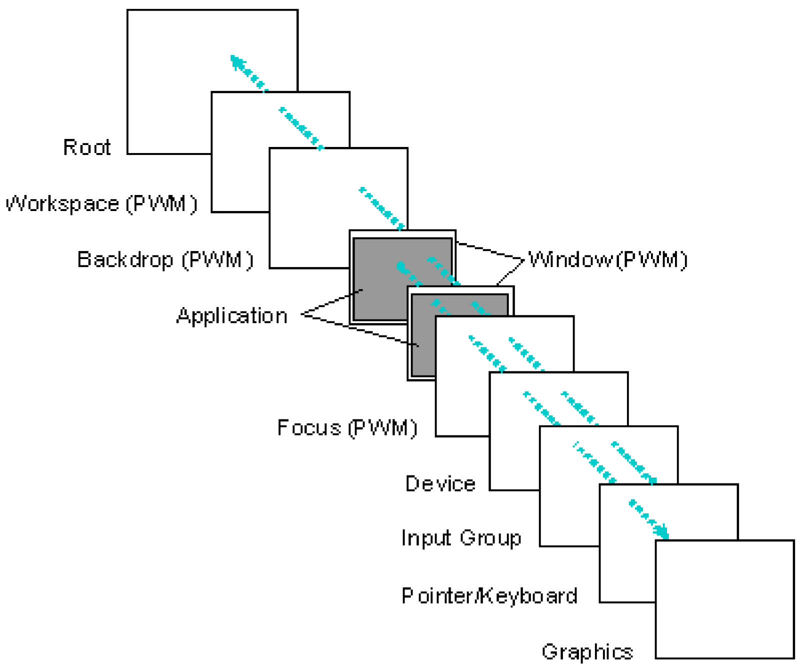
This abstraction is roughly parallel to the concept of a microkernel OS being incapable of filesystem or device I/O, but relying instead on external processes to provide these high-level services. Just as this allows a microkernel OS to scale up or down in size and functionality, so also a microkernel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

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

The Photon event space

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

Processes that provide device driver services place regions at the front of the event space. In addition to managing the event space and root region, the Photon microkernel maintains an on-screen pointer, projected as *draw events* towards the user.



Photon uses a series of regions, ranging from the Root region at the back of the Photon event space to the Graphics region at the front. Draw events start at an application's region and move forward to the Graphics region. Input events start at the Pointer/Keyboard region and travel towards the Root region.

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

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

The interaction between events and regions is the basis for the input and output facilities in Photon. Pen, mouse, and keyboard events travel away from the user towards the root plane, with the location of the on-screen pointer associated with them. Draw events originate from regions and travel towards the device plane and the user.

Regions

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

- *sensitivity* bitmask
- *opaque* bitmask.

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

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

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

Bitmask combination:	Description:
Not sensitive, transparent	The event passes through the region, unmodified, without the region owner being notified. The process owning the region simply isn't interested in the event.
Not sensitive, opaque	The event is clipped by the region as it passes through; the region owner isn't notified. For example, most applications would use this attribute combination for draw event clipping, so that an application's window wouldn't be overwritten by draw events coming from underlying windows.
Sensitive, transparent	A copy of the event will be sent to the region owner; the event will continue, unmodified, through the event space. A process wishing to log the flow of events through the event space could use this combination.
Sensitive, opaque	A copy of the event will be sent to the region owner; the event will also be clipped by the region as it passes through. By setting this bitmask combination, an application can act as an event filter or translator. For every event received, the application can process and regenerate it, arbitrarily transformed in some manner, possibly traveling in a new direction, and perhaps sourced from a new coordinate in the event space. For example, a region could absorb pen events, perform handwriting recognition on those events, and then generate the equivalent keystroke events.

Events

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

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

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

- keystrokes
- pen and mouse button actions

- pen and mouse motion
- region boundary crossings
- expose and covered events
- draw events
- drag events.

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

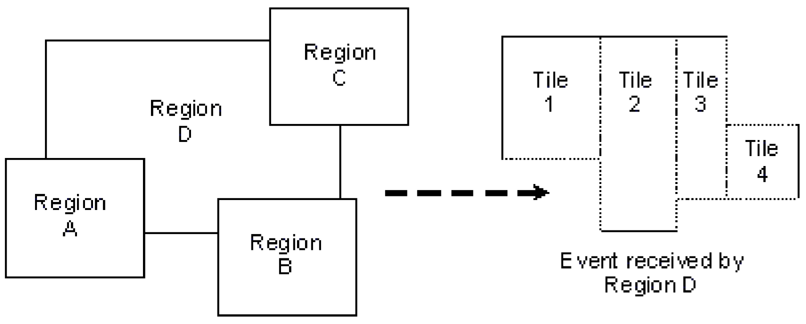
The rectangle list attached to the event can describe one or more rectangular regions, or it can be a "point-source" - a single rectangle where the upper-left corner is the same as the lower-right corner.

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

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

If the event goes through a region that clips the upper-left corner out of the draw event, the rectangle list is modified to contain only the two rectangles that would define the area remaining to be drawn. These resulting rectangles are called "tiles."

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



Regions opaque to draw events are clipped out, resulting in an area consisting of rectangular tiles.

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

Graphics drivers

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

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

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

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

Multiple graphics drivers

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

These multiple regions could be placed adjacent to each other, describing an array of "drawable" tiles, or overlapped in various ways. Since Photon inherits QNX's network transparency, Photon applications or drivers can run on any network node, allowing additional graphics drivers to extend the graphical space of Photon to the physical displays of many networked computers. By having the graphics driver regions overlap, the draw events can be replicated onto multiple display screens.

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

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

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

Color model

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

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

Scalable fonts

In addition to full support of bitmap fonts, Photon also provides *scalable* fonts. These fonts can be scaled to virtually any point size as well as anti-aliased (16 shades) for crisp, clean output on screen at practically any resolution.

Photon's scalable fonts are achieved by means of a high-speed font server that retrieves highly compressed font data from Portable Font Resource files (`*.pfr`), and then renders the font character shapes at any point size and resolution. Note that PFRs provide better than 2-to-1 compression over PostScript fonts.

Font sets

Core Latin set

The Photon *Core Latin* set (`latin1.pfr`), which comprises the Unicode *Basic Latin* (U+0000 - U+007F) and the *Latin-1 Supplement* (U+0080 - U+00FF) character sets, includes the following scalable fonts:

- Dutch
- Dutch Bold
- Dutch Italic
- Dutch Bold Italic
- Swiss
- Swiss Bold
- Swiss Italic
- Swiss Bold Italic
- Courier
- Courier Bold
- Courier Italic
- Courier Bold Italic

Extended Latin set

The Photon *Extended Latin* set (`latinx.pfr`) comprises the Unicode *Latin Extended-A* (U+0100 - U+017F) and *Latin Extended-B* (U+0180 - U+0217) sets. The Extended Latin set includes these fonts:

- Dutch
- Dutch Bold
- Dutch Italic (algorithmically generated)
- Dutch Bold Italic (algorithmically generated)
- Swiss
- Swiss Bold
- Swiss Italic (algorithmically generated)
- Swiss Bold Italic (algorithmically generated)

Languages supported

Armed with the Photon Core Latin set, the developer can support a host of languages, including:

Danish
Dutch
English
Finnish
Flemish
French
German
Hawaiian
Icelandic
Indonesian
Irish
Italian
Norwegian
Portuguese
Spanish
Swahili
Swedish

By adding the Photon Extended Latin set (`latinx.pfr`), the developer can support several additional languages, including:

Afrikaans
Basque
Catalan

Croatian
Czech
Esperanto
Estonian
Greenlandic
Hungarian
Latvian
Lithuanian
Maltese
Polish
Romanian
Slovak
Turkish
Welsh

International language supplements

Several language supplement packages are available for Photon:

- *Japanese Language Supplement* - offers developers full support for Japanese characters.

  Besides a complete set of PFRs (*Hon Mincho* Kanji set), the package also includes a front-end processor that allows users to provide input using either an English or a Japanese keyboard (via the popular VJE input method used in Japan).

- *Chinese Language Supplement* - offers developers full support for Chinese characters as well as a corresponding input method.
- *Korean Language Supplement* - offers developers full support for Korean characters as well as a corresponding input method.
- *Cyrillic Language Supplement* - offers developers full support for Russian or any language that relies on the Cyrillic alphabet. This package also includes the appropriate keyboard drivers to support English or Cyrillic keyboards.

Unicode multilingual support

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

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

UTF-8 encoding

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

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

- Unicode characters from U+0000 to U+007E (ASCII set) map to UTF-8 bytes 00 to 7E (ASCII values).
- ASCII values don't otherwise occur in a UTF-8 transformation, giving complete compatibility with historical filesystems that parse for ASCII bytes.
- UTF-8 simplifies conversions to and from Unicode text.
- The first byte indicates the number of bytes to follow in a multibyte sequence, allowing for efficient forward parsing.
- Finding the start of a character from an arbitrary location in a byte stream is efficient, because you need to search at most four bytes backwards to find an easily recognizable initial byte. For example:

```
isInitialByte = ((byte & 0xC0) != 0x80);
```

- UTF-8 is reasonably compact in terms of the number of bytes used for encoding.

The OS library includes a convenient set of conversion functions:

Function	Description
mblen()	Length of a multibyte string in characters
mbtowc()	Convert multibyte character to wide character
mbstowcs()	Convert multibyte string to wide character string
wctomb()	Convert wide character to its multibyte representation
wcstombs()	Convert wide character string to multibyte string

In addition to the conversion functions listed above, developers can also rely on Photon's own *PxTranslate* library functions, which will translate various character set encodings to and from UTF-8.

Animation support

Photon provides flicker-free animations via a special "double-buffer" container widget (`PtDBContainer`) that creates a dedicated off-screen memory context for drawing images.

The `PtDBContainer` widget uses a block of shared memory large enough to hold an image the size of its canvas, and will also dynamically load the render shared library.

Printing support

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

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

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

The Photon Window Manager

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

The window manager is built on the concept of filtering events with additional regions placed behind application regions, upon which a title bar, resize handles, and other "gadgets" are drawn and interacted with. Since the replaceable window manager implements the actual "look and feel" of the environment, various UI flavors can be optionally installed.

Widget library

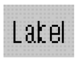
Photon provides a library of components known as *widgets* - objects that can manage much of their on-screen behavior automatically, without explicit programming effort. As a result, a complete application can be quickly assembled by combining widgets in various ways and then attaching C code to the appropriate callbacks the widgets provide. The Photon Application Builder (PhAB), which is included as part of the Photon development system, provides an extensive widget palette in its visual development environment.

Photon provides a wide range of widgets:

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

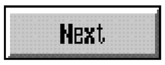
Basic widgets

Label widget (`PtLabel`)



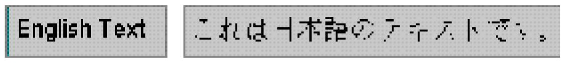
The label widget is mainly used to display textual information. The `PtLabel` widget is the superclass for all text-based widgets, providing many customizable attributes (e.g. font typeface, pop-up balloons, colors, borders, alignment, margins, etc.), all of which are inherited by all its subclasses.

Push button widget (`PtButton`)



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

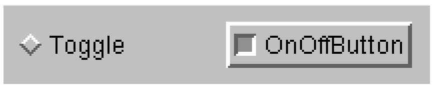
Text input widgets (`PtText`, `PtMultiText`)



Photon provides two text-input widgets:

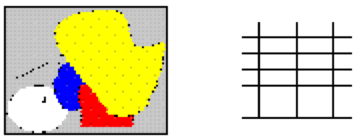
- a simple single-line input widget (`PtText`) commonly used in forms
- a powerful wordprocessor-like multi-line widget (`PtMultiText`) providing full editing capabilities, word wrapping, automatic scrolling, and multi-font line segments.

Toggle button widgets (`PtToggleButton`, `PtOnOffButton`)



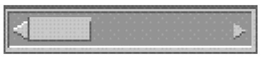
Toggle buttons are objects that display two states - either on or off. Photon provides two different types of toggle buttons, each with a different visual appearance. Toggle buttons are used to display or request state information related to a command or action about to be performed.

Graphical widgets (`PtArc`, `PtPixel`, `PtRectangle`, `PtLine`, `PtPolygon`, `PtEllipse`, `PtBezier`, `PtGrid`)



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

Scrollbar widget (`PtScrollbar`)

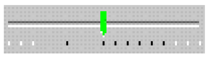


A scrollbar widget is used to scroll the display of a viewable area. The scrollbar is combined with other widgets (e.g. `PtList`, `PtScrollArea`) to allow scrolling.

Separator widget (`PtSeparator`)

The separator widget is used to separate two or more different areas, giving a better visual appearance. The separator can be customized for many different styles and looks.

Slider widget (`PtSlider`)

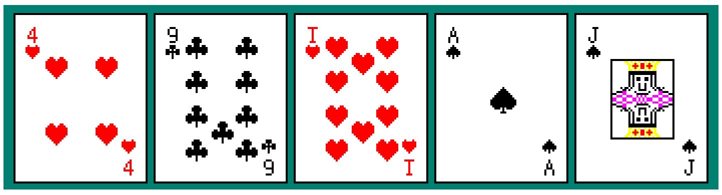


Sliders are different from scrollbars. A scrollbar defines a range, whereas a slider defines a single value. The slider widget provides a rich list of customizable attributes.

Timer widget (`PtTimer`)

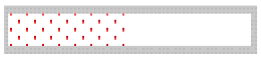
The timer widget makes using timers a snap. This widget has no visual appearance - it simply defines a callback whenever a timer event is triggered. The application sets the timer value and optional repeat value. When the timer goes off, the application is notified.

Graphic image widgets (`PtBitmap`, `PtLabel`, `PtButton`)



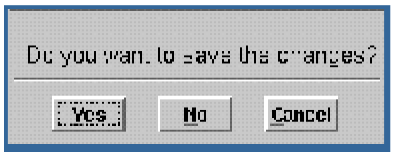
Photon supports every major graphic file standard, so you can import graphics and display them inside widgets. Many Photon widgets directly support displaying graphics - the most common are `PtButton` for making push-button toolbars and `PtLabel` for displaying images.

Progress bar widget (`RtProgress`)



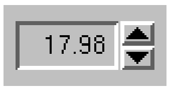
If an application needs to do something that takes a fair amount of time (e.g. loading a file), it can use the progress bar widget to let the user know what's happening and, more importantly, how much longer the process is going to take. The progress bar can be horizontal or vertical and has many attributes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

Message widget (`PtMessage`)



Pop-up messages and notifications are quite common in a windowing environment. Photon provides a very handy message dialog widget that displays a message and up to 3 user-response buttons. There's also a very useful modal dialog function call (*PtAskQuestion()*) based on the message widget.

Numeric widgets (`PtNumericInteger`, `PtNumericFloat`)

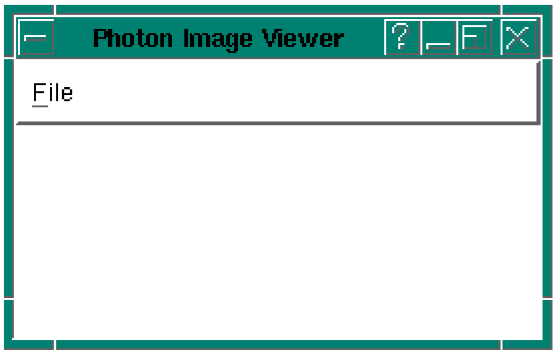


The `PtNumericInteger` class lets the user specify integer values between given minimum and maximum values. The `PtNumericFloat` class lets the user enter floating-point values.

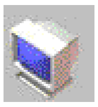
A `PtUpDown` widget is also shown to allow the user to increase or decrease the value by a set amount.

Container widgets

Window and icon widgets (`PtWindow`, `PtIcon`)

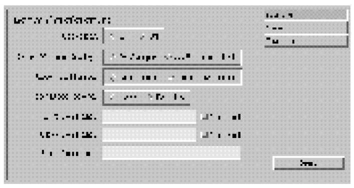


Windows are the main application containers. The main UI components (menu bars, toolbars, etc.) appear with the window widget. The widget automatically handles all the necessary interactions with the Photon Window Manager (PWM) - all you need to specify is what should and shouldn't be rendered or managed.



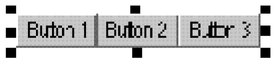
Icon widgets are closely associated with windows and are displayed in the Photon Desktop Manager launch folders and PWM taskbar.

Bulletin board widgets (`PtPane`)



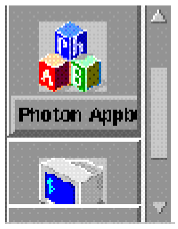
Bulletin board widgets are simple container widgets that are used to hold other widgets. Although these are parent widgets, they don't manage their child widgets in any way. They are quite useful for designing form layouts commonly found in dialog windows.

Group widget (`PtGroup`)



The group widget is a very powerful widget that manages the geometry of all its child widgets. It can align the widgets horizontally, vertically, or in a matrix. The group widget can be anchored to the side of any other container (like a window) so that it automatically resizes when the window resizes. The group widget also provides attributes that let you specify whether the children should be stretched to fit the group if it's resized larger due to anchoring.

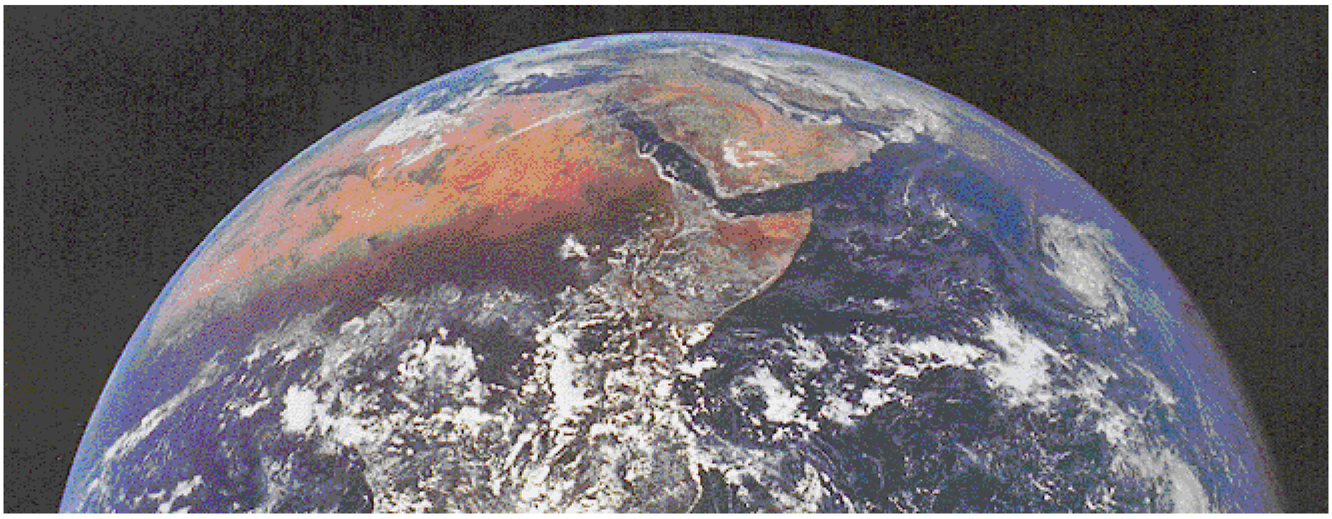
Scrolling area widget (`PtScrollArea`)



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

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

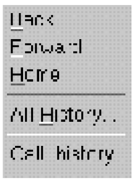
Background widget (`PtBkgd`)



The background widget provides a way to create fancy background displays, from simple color gradations to tiled textures. Just about any background requirement is handled by this widget.

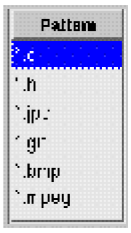
Advanced widgets

Menu related widgets (`PtMenu`, `PtMenuBar`, `PtMenuButton`)



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

List widget (`PtList`)

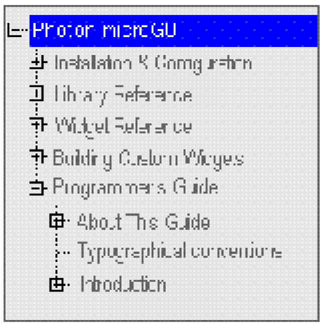


The list widget is a very powerful widget that manages a list of items. It provides many different selection modes, including single selection, multiple selection and range selection. The list widget also supports multi-columned lists through the use of a divider widget (`PtDivider`).

Pulldown list widget (`PtComboBox`)

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

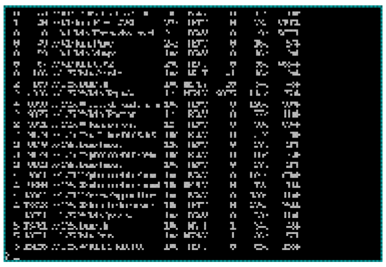
Tree widget (`PtTree`)



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

Photon applications that use the tree widget include: the File Manager (directory display), PhAB (widget layout), `vsin` (process list), and many others.

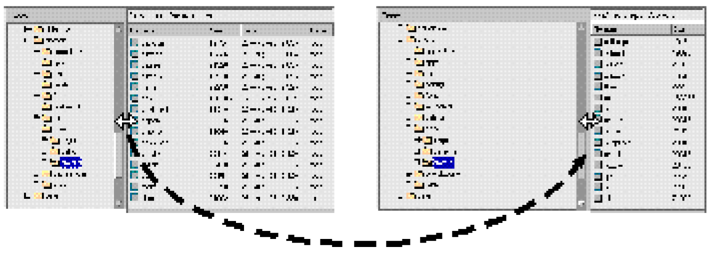
Terminal widgets (`PtTty`, `PtTerminal`)



Imagine having a text console inside your application. That's exactly what this widget does. It creates and manages an entire text-mode terminal inside a widget. Just drop it into your application and you've created your very own `pterm` (our terminal application).

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

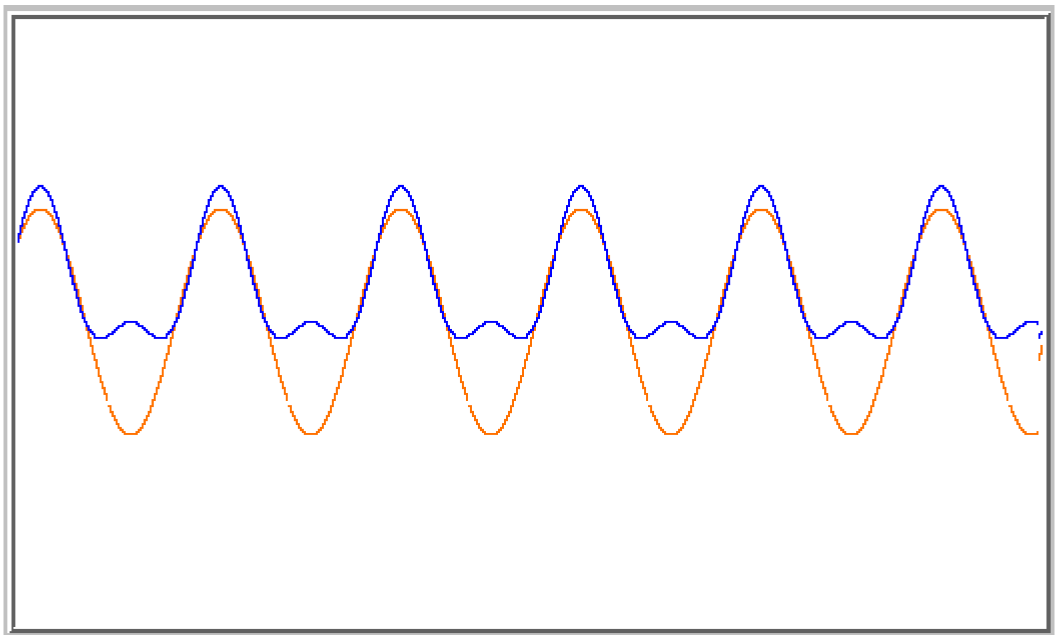
Divider widget (`PtDivider`)



This powerful widget manages its children in a unique and useful way. When you place two or more widgets inside a divider widget, it automatically puts little separators in between the child widgets. Using these separators, the user can drag back and forth, causing the child widgets on either side of the separator to be resized. This is very useful for creating resizable column headings for lists. In fact, if you drop a divider widget into a list widget, it will automatically turn your simple list into a resizable multi-column list.

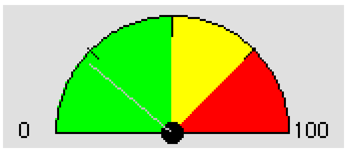
Dividers aren't limited to just labels or buttons. Any widgets can be placed inside to create side-by-side resizable trees, scroll areas, and so on.

Trend graph widget (`RtTrend`)



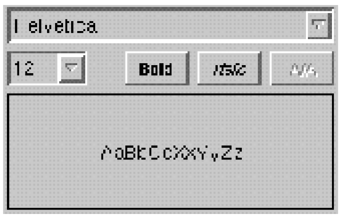
Realtime systems often require trend graphs. Photon comes with a trend bar widget that supports the display of multiple trend lines simultaneously. If your graphics hardware supports masked blits, it can even smooth-scroll the trend across grid lines.

Realtime meter widget (`RtMeter`)



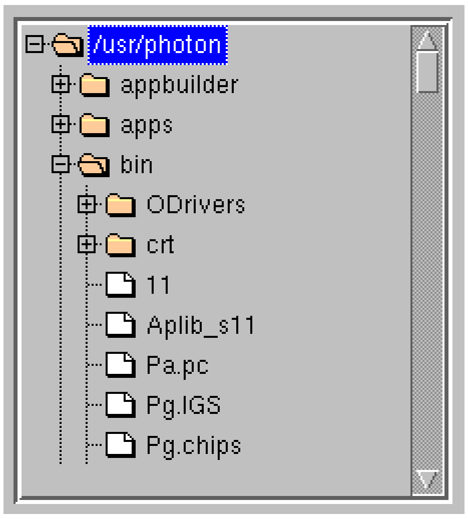
The `RtMeter` widget is drawn as a half circle with divisional ticks at 1/3, 1/2, and 2/3 of the arc. The needle can be moved with the mouse, the keyboard, or programmatically. A single mouse click moves the meter to the current mouse position; a mouse "drag" causes the needle to follow the mouse through meter values.

Font selection dialog (`PtFontSel`)



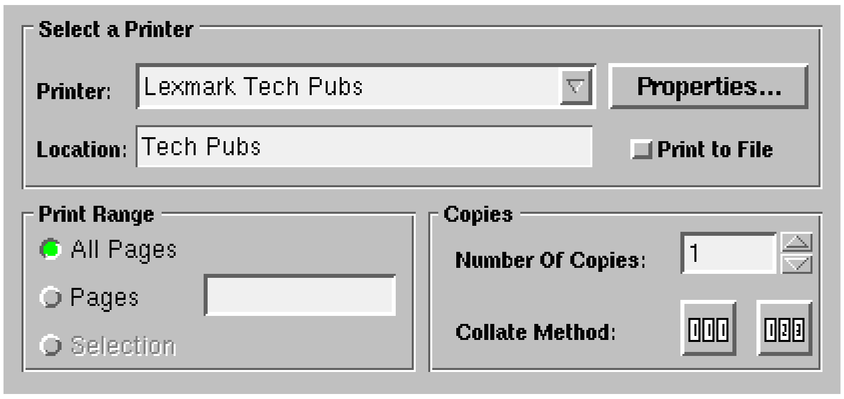
To accommodate the wide selection of fonts available for Photon, a Font Selection widget is provided. This widget can read the standard font-mapping files and display a list of all available fonts. It allows you to choose the typeface, style (bold, italic, etc.) and also indicate whether the font should be anti-aliased.

File selection widget (`PtFileSel`)



The `PtFileSel` widget is a tree widget that displays files, directories, links to files or directories, or custom entries. Besides selecting a particular file in response to an application prompt, users can also use this widget to navigate an entire filesystem and choose their own file and directory.

Print selection dialog (`PtPrintSel`)



Select a Printer

Printer: Lexmark Tech Pubs

Location: Tech Pubs

Print to File

Print Range: [✓ All Pages] Pages Selection

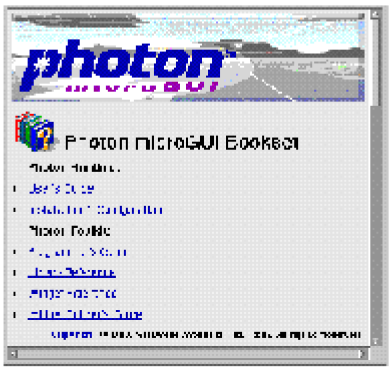
Copies

Number Of Copies: 1

Collate Method:

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

HTML viewer widget (`PtHtml`)



Using the HTML widget makes it easy to create a customized help viewer. It will format a standard HTML file and even autoload all the images. It handles resizing, scrolling, just about everything you would need to do. This widget is ideal for creating helpviewers for touchscreen environments.

Widget building toolkit

If all the standard Photon widgets aren't enough, you can easily build your own! The Photon Development System comes with complete documentation and sample source code to create your own custom widgets. You can sub-class off of existing widgets to inherit their functionality or create a complete new widget class tree. Your widget possibilities are endless.

Summary

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

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