About This Guide

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

The following table may help you find information quickly:

To find out about:	**Go to:**
OS design goals; message-passing IPC	The Philosophy of the QNX OS
System services	The QNX OS Microkernel
Sharing information between processes	Interprocess Communication (IPC)
System event monitoring	The Microkernel Instrumentation
Memory management, pathname management, etc.	Process Manager
Shared objects	Dynamic Linking
Device drivers	Resource Managers
Power-Safe, QNX compressed, QNX Trusted Disk, Image, DOS, Ext2, and other filesystems	Filesystems
Serial and parallel devices	Character I/O
Network subsystem	Networking Architecture
TCP/IP implementation	TCP/IP Networking
Fault recovery	High Availability
An overview of hard and soft real time	What is Real Time and Why Do I Need It?
Terms used in QNX OS documentation	Glossary

For information about programming, see *Getting Started with the QNX OS* and the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

Character I/O

QNX SDP | 8.0 | System Architecture | Developer | User

A key requirement of any realtime operating system is high-performance character I/O.

Character devices can be described as devices to which I/O consists of a sequence of bytes transferred serially, as opposed to block-oriented devices (e.g., disk drives).

As in the POSIX and UNIX tradition, these character devices are located in the OS pathname space under the **/dev** directory. For example, a serial port to which a modem or terminal could be connected might appear in the system as: **/dev/ser1**

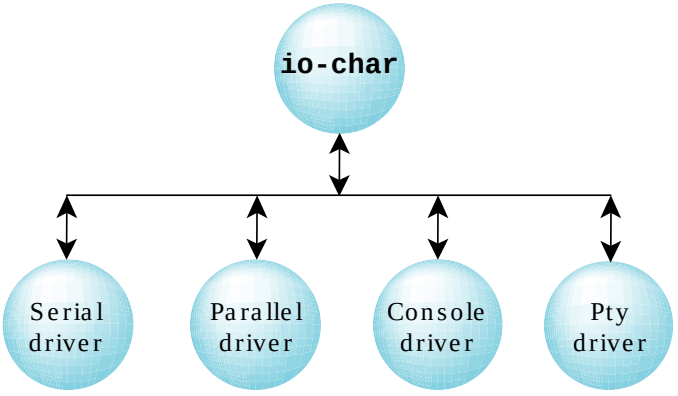
Typical character devices found on PC hardware include:

- serial ports
- parallel ports
- text-mode consoles
- pseudo terminals (ptys)

Programs access character devices using the standard *open()*, *close()*, *read()*, and *write()* API functions. Additional functions are available for manipulating other aspects of the character device, such as baud rate, parity, flow control, etc.

Since it's common to run multiple character devices, they have been designed as a family of drivers and a library called `io-char` to maximize code reuse.

Figure 1The `io-char` library is used by multiple drivers.



The `io-char` module contains all the code to support POSIX semantics on the device. It also contains a significant amount of code to implement character I/O features beyond POSIX but desirable in a realtime system. Since this code is in the common library, all drivers inherit these capabilities.

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

Once a single character device is running, the memory cost of adding more devices is minimal, since only the code to implement the new driver structure would be new.

Page updated: August 11, 2025

Console devices

QNX SDP | 8.0 | System Architecture | Developer | User

System consoles (with VGA-compatible graphics chips in *text* mode) are managed by the `devc-con` or `devc-con-hid` driver. The video display card/screen and the system keyboard are collectively referred to as the *physical console*.

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

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

Terminal emulation

The console drivers emulate an ANSI terminal.

Page updated: August 11, 2025

Device control

QNX SDP | 8.0 | System Architecture | Developer | User

Low-level device control is implemented using the *devctl()* call.

The POSIX terminal control functions are layered on top of *devctl()* as follows:

tcgetattr()
: Get terminal attributes.

tcsetattr()
: Set terminal attributes.

tcgetpgrp()
: Get ID of process group leader for a terminal.

tcsetpgrp()
: Set ID of process group leader for a terminal.

tcsendbreak()
: Send a break condition.

tcflow()
: Suspend or restart data transmission/reception.

QNX OS extensions

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

tcdropline()
: Initiate a disconnect. For a serial device, this will pulse the DTR line.

tcinject()
: Inject characters into the canonical buffer.

The `io-char` module acts directly on a common set of *devctl()* commands supported by most drivers.
Applications send device-specific *devctl()* commands through `io-char` to the drivers.

Page updated: August 11, 2025

Input modes

QNX SDP | 8.0 | System Architecture | Developer | User

Each device can be in a *raw* or *edited* input mode.

Raw input mode

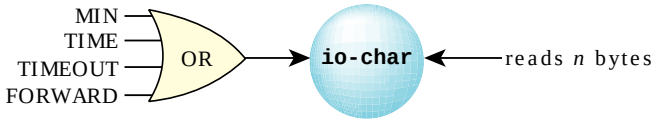
In raw mode, `io-char` performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data.

Fullscreen programs and serial communications programs are examples of applications that use a character device in raw mode.

In raw mode, the hardware-handling code receives each character into the raw input buffer. When an application requests data from the device, it can specify under what conditions an input request is to be satisfied. Until the conditions are satisfied, the driver won't process the incoming data (beyond storing it), and the driver won't return any data to the application. The normal case of a simple read by an application would block until at least one character was available.

The following diagram shows the full set of available conditions:

Figure 1Conditions for satisfying an input request.



MIN	Respond when at least this number of characters arrives.
TIME	Respond if a pause in the character stream occurs.
TIMEOUT	Respond if an overall amount of time passes.
FORWARD	Respond if a framing character arrives.

In the case where multiple conditions are specified, the read will be satisfied when any one of them is satisfied.

MIN

The qualifier *MIN* is useful when an application has knowledge of the number of characters it expects to receive.

Any protocol that knows the character count for a frame of data can use *MIN* to wait for the entire frame to arrive. This significantly reduces IPC and process scheduling. *MIN* is often used in conjunction with *TIME* or *TIMEOUT*. *MIN* is part of the POSIX standard.

TIME

The qualifier *TIME* is useful when an application is receiving streaming data and wishes to be notified when the data stops or pauses. The pause time is specified in 1/10ths of a second. *TIME* is part of the POSIX standard.

TIMEOUT

The qualifier *TIMEOUT* is useful when an application has knowledge of how long it should wait for data before timing out. The timeout is specified in 1/10ths of a second.

Any protocol that knows the character count for a frame of data it expects to receive can use *TIMEOUT*. This in combination with the baud rate allows a reasonable guess to be made when data should be available. It acts as a deadman timer to detect dropped characters. It can also be used in interactive programs with user input to time out a read if no response is available within a given time.

TIMEOUT is a QNX OS extension and is not part of the POSIX standard.

FORWARD

The qualifier *FORWARD* is useful when a protocol is delimited by a special framing character. For example, the PPP protocol used for TCP/IP over a serial link starts and ends its packets with a framing character. When used in conjunction with *TIMEOUT*, the *FORWARD* character can greatly improve the efficiency of a protocol implementation. The protocol process will receive complete frames, rather than character by character. In the case of a dropped framing character, *TIMEOUT* or *TIME* can be used to quickly recover.

This minimizes the amount of IPC work for the OS and results in a much lower processor utilization for a given TCP/IP data rate. It's interesting to note that PPP doesn't contain a character count for its frames. Without the data-forwarding character, an implementation might be forced to read the data one character at a time.

FORWARD is a QNX OS extension and is not part of the POSIX standard.

The ability to “push” the processing for application notification into the service-providing components of the OS reduces the frequency with which user-level processing must occur. This minimizes the IPC work to be done in the system and frees CPU cycles for application processing. In addition, if the application implementing the protocol is executing on a different network node than the communications port, the number of network transactions is also minimized.

For intelligent, multiport serial cards, the data-forwarding character recognition can also be implemented within the intelligent serial card itself, thereby significantly reducing the number of times the card must interrupt the host processor for interrupt servicing.

Edited input mode

In edited mode, `io-char` performs line-editing operations on each received character. Only when a line is “completely entered”—typically when a carriage return (CR) is received—will the line of data be made available to application processes. This mode of operation is often referred to as *canonical* or sometimes “cooked” mode.

Most non-full-screen applications run in edited mode, because this allows them to deal with the data a line at a time rather than have to examine each character received, scanning for an end-of-line character. In this mode, the raw input buffer receives each character through the hardware handling code. Unlike raw mode where the driver only processes data when the input request satisfies the conditions, the driver processes each byte as it arrives.

When the driver runs, code in `io-char` will examine the character and apply it to the canonical buffer in which it's building a line. When a line is complete and an application requests input, the line will be transferred from the canonical buffer to the application—the transfer is direct from the canonical buffer to the application buffer without any intervening copies.

The editing code correctly handles multiple pending input lines in the canonical buffer and allows partial lines to be read. This can happen, for example, if an application asked only for 1 character when a 10-character line was available. In this case, the next read will continue where the last one left off.

The `io-char` module provides a rich set of editing capabilities, including full support for moving over the line with cursor keys and for changing, inserting, or deleting characters. Here are some of the more common capabilities:

LEFT
: Move the cursor one character to the left.

RIGHT
: Move the cursor one character to the right.

HOME
: Move the cursor to the beginning of the line.

END
: Move the cursor to the end of the line.

ERASE
: Erase the character to the left of the cursor.

DEL
: Erase the character at the current cursor position.

KILL
: Erase the entire input line.

UP
: Erase the current line and recall a previous line.

DOWN
: Erase the current line and recall the next line.

INS
: Toggle between insert mode and typeover mode (every new line starts in insert mode).

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

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

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

Pseudo terminals (ptys)

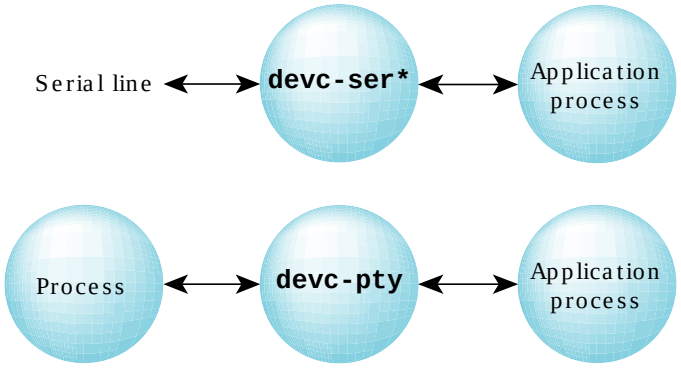
QNX SDP | 8.0 | System Architecture | Developer | User

Pseudo terminals are managed by the `devc-pty` driver.

Command-line arguments to `devc-pty` specify the number of pseudo terminals to create.

A pseudo terminal (pty) is a *pair* of character devices: a master device and a worker device. The worker device provides an interface identical to that of a tty device as defined by POSIX. However, while other tty devices represent hardware devices, the worker device instead has another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the worker device as input; anything written on the worker device is presented as input to the master device. As a result, pseudo-ttys can be used to connect processes that would otherwise expect to be communicating with a character device.

Figure 1Pseudo-ttys.



Ptys are routinely used to create pseudo-terminal interfaces for programs, which uses TCP/IP to provide a terminal session to a remote system.

Page updated: August 11, 2025

Serial devices

QNX SDP | 8.0 | System Architecture | Developer | User

Serial communication channels are managed by the `devc-ser*` family of driver processes. These drivers can manage more than one physical channel and provide character devices with names such as **/dev/ser1**, **/dev/ser2**, etc.

When `devc-ser*` is started, command-line arguments can specify which—and how many—serial ports are installed. On a PC-compatible system, this will typically be the two standard serial ports often referred to as **com1** and **com2**. The `devc-ser*` driver directly supports most nonintelligent multiport serial cards.

QNX OS includes various serial drivers (e.g., `devc-ser8250`). For details, see the `devc-ser*` entries in the *Utilities Reference*.

The `devc-ser*` drivers support hardware flow control (except under edited mode) provided that the hardware supports it. Loss of carrier on a modem can be programmed to deliver a SIGHUP signal to an application process (as defined by POSIX).

Page updated: August 11, 2025

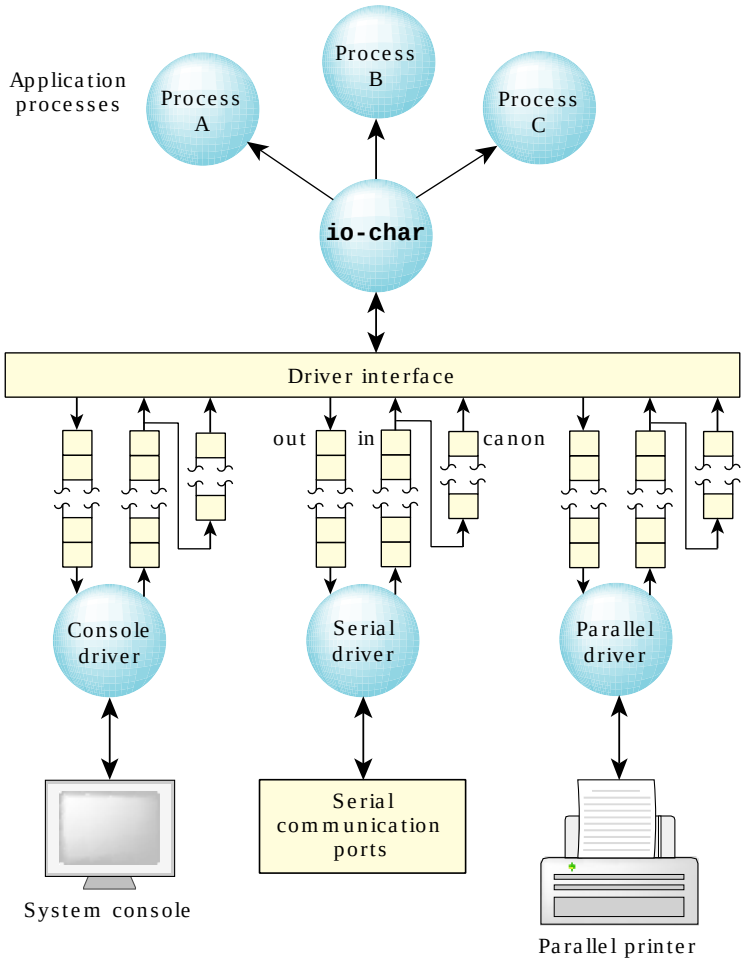
Driver/`io-char` communication

QNX SDP | 8.0 | System Architecture | Developer | User

The `io-char` library manages the flow of data between an application and the device driver. Data flows between `io-char` and the driver through a set of memory queues associated with each character device.

Three queues are used for each device. Each queue is implemented using a first-in, first-out (FIFO) mechanism.

Figure 1Device I/O in the QNX OS.



Received data is placed into the *raw* input queue by the driver and is consumed by `io-char` only when application processes request data. (For details on raw versus edited or canonical input, see the section "Input modes" later in this chapter.)

The `io-char` module places output data into the output queue to be consumed by the driver as characters are physically transmitted to the device. The module calls a trusted routine within the driver each time new data is added so it can "kick" the driver into operation (in the event that it was idle). Since output queues are used, `io-char` implements *write-behind* for all character devices. Only when the output buffers are full will `io-char` cause a process to block while writing.

The canonical queue is managed entirely by `io-char` and is used while processing input data in *edited* mode. The size of this queue determines the maximum edited input line that can be processed for a particular device.

The sizes of these queues are configurable using command-line options. Default values are usually more than adequate to handle most hardware configurations, but you can "tune" these to reduce overall system memory requirements, to accommodate unusual hardware situations, or to handle unique protocol requirements.

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

Page updated: August 11, 2025

Copyright and patent notice

QNX SDP | 8.0 | System Architecture | Developer | User

BlackBerry Limited
2200 University Avenue E.
Waterloo, Ontario
N2K 0A7
Canada

Voice: +1 519 888-7465
Fax: +1 519 888-6906
Email: info@qnx.com
Web: https://www.qnx.com/

August 11, 2025

Page updated: August 11, 2025

Dynamic Linking

QNX SDP | 8.0 | System Architecture | Developer | User

In a typical system, a number of programs will be running. Each program relies on a number of functions, some of which will be "standard" C library functions, like *printf()*, *malloc()*, *write()*, etc.

If every program uses the standard C library, it follows that each program would normally have a unique copy of this particular library present within it. Unfortunately, this results in wasted resources. Since the C library is common, it makes more sense to have each program *reference* the common instance of that library, instead of having each program *contain* a copy of the library. This approach yields several advantages, not the least of which is the savings in terms of total system memory required.

Before we go any further, let's look at some terminology:

Linker
: A tool, such as `ld`, that you typically run just after compiling your program, in order to combine object and archive files, relocate their data, and resolve symbol references.

Runtime linker
: A tool that finds and loads shared objects when you run your program. This is also known as a *dynamic linker*, but we'll use *runtime linker* to avoid any confusion with dynamic linking, which the (non-runtime) linker does.

  The runtime linker's name is `ldqnx`. In the `.interp` section of an ELF file, it's called **/usr/lib/ldqnx-64.so**. You'll need to include this in your OS image; see the entry for `mkifs` in the *Utilities Reference* for more details.

Statically linked
: The program and the particular library that it references are combined by the linker at link time.

  This means that the binding between the program and the particular library is fixed and known at link time—well in advance of the program's ever running. It also means that we can't change this binding, unless we relink the program with a new version of the library.

  You might consider linking a program statically if you aren't sure whether the correct version of a library will be available at runtime, or if you are testing a new library version that you don't yet want to install as shared.

  Programs that are linked statically are linked against archives of objects (*libraries*) that typically have the extension of **.a**. An example of such a collection of objects is the standard C library, **libc.a**.

Dynamically linked
: The program and the particular library that it references *aren't* combined by the linker at link time.

  Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references. This means that the binding between the program and the shared object is done *at runtime*—before the program starts, the appropriate shared objects are found and bound.

  This type of program is called a *partially bound executable*, because it isn't fully *resolved*—the linker, at link time, didn't associate all referenced symbols in the program with specific code from the library. Instead, the linker simply said: "This program calls some functions within a particular shared object, so I'll just make a note of *which* shared object these functions are in, and continue on." Effectively, this defers the binding until runtime.

  Programs that are linked dynamically are linked against shared objects that have the extension **.so**. An example of such an object is the shared object version of the standard C library, **libc.so**.

  You use a command-line option to the compiler driver `qcc` to tell the toolchain whether you're linking statically or dynamically. This command-line option then determines the extension used (either **.a** or **.so**).

Augmenting code at runtime

Taking this one step further, a program may not know which functions it needs to call until it's running. While this may *seem* a little strange initially (after all, how could a program *not* know what functions it's going to call?), it really can be a very powerful feature. Here's why.

Consider a "generic" disk driver. It starts, probes the hardware, and detects a hard disk. The driver would then dynamically load the **io-blk** code to handle the disk blocks, because it found a block-oriented device. Now that the driver has access to the disk at the block level, it finds two partitions present on the disk: a DOS partition and a Power-Safe partition. Rather than force the disk driver to contain filesystem drivers for all possible partition types

it may encounter, we kept it simple: it doesn't have *any* filesystem drivers! At runtime, it detects the two partitions and *then* knows that it should load the `fs-dos.so` and `fs-qnx6.so` filesystem code to handle those partitions.

By deferring the decision of which functions to call, we've enhanced the flexibility of the disk driver (and also reduced its size).

Page updated: August 11, 2025

How shared objects are used

QNX SDP | 8.0 | System Architecture | Developer | User

To understand how a program makes use of shared objects, let's first see the format of an executable and then examine the steps that occur when the program starts.

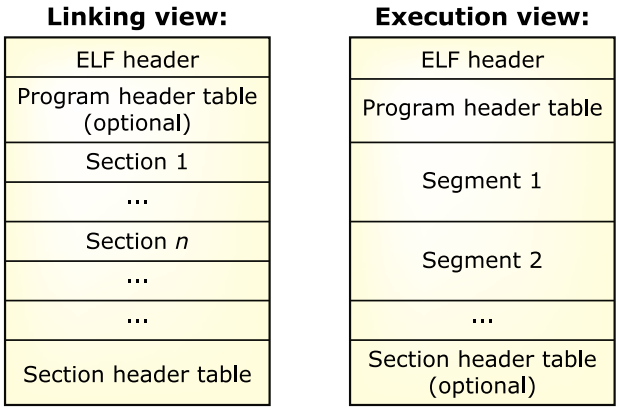
ELF format

The QNX OS uses the ELF (Executable and Linking Format) binary format. ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at runtime.

In the following diagram, we show two views of an ELF file: the linking view and the execution view. The linking view, which is used when the program or library is linked, deals with *sections* within an object file. Sections contain the bulk of the object file information: data, instructions, relocation information, symbols, debugging information, etc. The execution view, which is used when the program runs, deals with *segments*.

At link time, the program or library is built by merging together sections with similar attributes into segments. Typically, all the executable and read-only data sections are combined into a single `text` segment, while the data and “BSS”s are combined into the `data` segment. These segments are called *load segments*, because they need to be loaded in memory at process creation. Other sections such as symbol information and debugging sections are merged into other, nonload segments.

Figure 1Object file format: linking view and execution view.



ELF without COFF

Most implementations of ELF loaders are derived from *COFF* (Common Object File Format) loaders; they use the linking view of the ELF objects at load time. This is inefficient because the program loader must load the executable using sections. A typical program could contain a large number of sections, each of which would have to be located in the program and loaded into memory separately.

QNX OS, however, doesn't rely at all on the COFF technique of loading sections. When developing our ELF implementation, we worked directly from the ELF specification and kept efficiency paramount. The ELF loader uses the “execution view” of the program. By doing so, the loader's task is greatly simplified: all it has to do is copy to memory the load segments (usually two) of the program or library. As a result, process creation and library loading operations are much faster.

Page updated: August 11, 2025

Loading a shared library at runtime

QNX SDP | 8.0 | System Architecture | Developer | User

A process can load a shared library at runtime by using the *dlopen()* call, which instructs the runtime linker to load this library. Once the library is loaded, the program can call any function within that library by using the *dlsym()* call to determine its address.

NOTE:
Remember: shared libraries are available only to processes that are *dynamically linked*.

The program can also determine the symbol associated with a given address by using the *dladdr()* call. Finally, when the process no longer needs the shared library, it can call *dlclose()* to unload the library from memory.

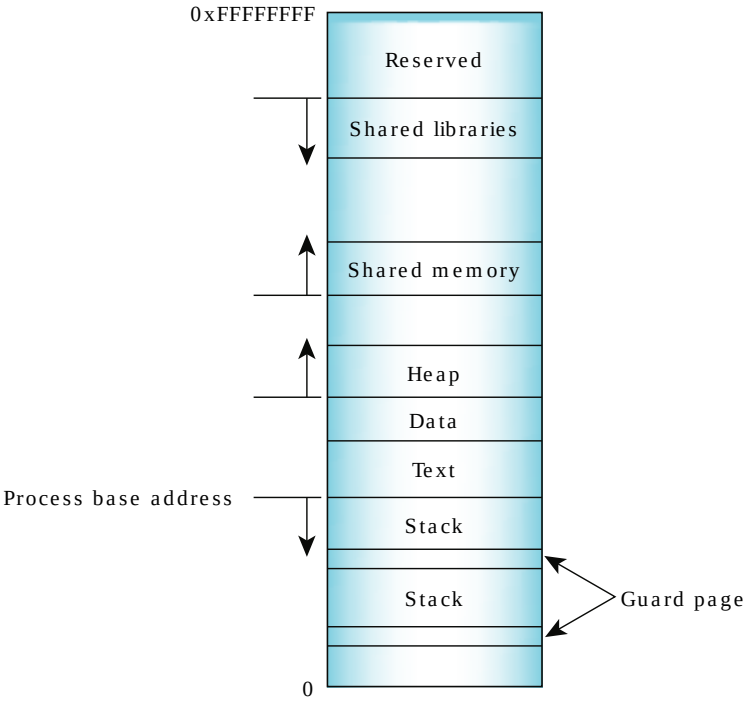
Page updated: August 11, 2025

Memory layout for a typical process

QNX SDP | 8.0 | System Architecture | Developer | User

The diagram below shows the memory layout of a typical process. The process load segments (corresponding to `text` and `data` in the diagram) are loaded at the process's base address. The main stack is located just below and grows downwards. Any additional threads that are created will have their own stacks, located below the main stack. Each of the stacks is separated by a guard page to detect stack overflows. The heap is located above the process load segments and grows upwards.

Figure 1Process memory layout on an x86 system.



In the middle of the process's address space, a large region is reserved for shared objects. Shared libraries are located at the top of the address space and grow downwards.

When a new process is created, the process manager first maps the two segments from the executable into memory. It then decodes the program's ELF header. If the program header indicates that the executable was linked against a shared library, the process manager will extract the name of the *dynamic interpreter* from the program header. The dynamic interpreter points to a shared library that contains the *runtime linker* code. The process manager will load this shared library in memory and will then pass control to the runtime linker code in this library.

Page updated: August 11, 2025

Runtime linker

QNX SDP | 8.0 | System Architecture | Developer | User

The runtime linker is invoked when a program that was linked against a shared object is started or when a program requests that a shared object be dynamically loaded. This linker is contained within the **ldqnx-64.so** library, which is separate from `libc`.

The runtime linker performs several tasks when loading a shared library (**.so** file):

1. If the requested shared library isn't already loaded in memory, the runtime linker loads it:
    - If the shared library name is fully qualified (i.e., begins with a slash), it's loaded directly from the specified location. If it can't be found there, no further searches are performed.
    - If it's not a fully qualified pathname, the linker searches for it as follows:
        a. If the executable's dynamic section contains a `DT_RPATH` tag, then the path specified by `DT_RPATH` is searched.
        b. If the shared library isn't found, the runtime linker searches for it in the directories specified by *LD_LIBRARY_PATH*.

        NOTE:
        For security reasons, the runtime linker unsets *LD_LIBRARY_PATH* if the binary has the setuid bit set.

        c. If the shared library still isn't found, then the linker searches for the default library search path as specified by the *LD_LIBRARY_PATH* environment variable to `procnto` (i.e., the CS_LIBPATH configuration string). If none has been specified, then the default library path is set to the image filesystem's path.
2. Once the requested shared library is found, it's loaded into memory. For ELF shared libraries, this is a very efficient operation: the runtime linker simply needs to use the *mmap()* call twice to map the two load segments into memory.
3. The shared library is then added to the internal list of all libraries that the process has loaded. The runtime linker maintains this list.
4. The runtime linker then decodes the dynamic section of the shared object.

This dynamic section provides information to the linker about other libraries that this library was linked against. It also gives information about the relocations that need to be applied and the external symbols that need to be resolved. The runtime linker will first load any other required shared libraries (which may themselves reference other shared libraries). It will then process the relocations for each library. Some of these relocations are local to the library, while others require the runtime linker to resolve a global symbol. In the latter case, the linker will search through the list of libraries for this symbol. In ELF files, hash tables are used for the symbol lookup, so they're very fast. The order in which libraries are searched for symbols is very important, as we'll see in the section on "Symbol name resolution" below.

Once all relocations have been applied, any initialization functions that have been registered in the shared library's init section are called. This is used in some implementations of C++ to call global constructors.

Page updated: August 11, 2025

Symbol name resolution

QNX SDP | 8.0 | System Architecture | Developer | User

When the runtime linker loads a shared library, the symbols within that library have to be resolved. The order and the scope of the symbol resolution are important. If a shared library calls a function that happens to exist by the same name in several libraries that the program has loaded, the order in which these libraries are searched for this symbol is critical. This is why the OS defines several options that can be used when loading libraries.

All the objects (executables and libraries) that have global scope are stored on an internal list (the *global list*). Any global-scope object, by default, makes available all of its symbols to any shared library that gets loaded. The global list initially contains the executable and any libraries that are loaded at the program's startup.

By default, when a new shared library is loaded by using the *dlopen()* call, symbols within that library are resolved by searching in this order through:

1. the list of libraries specified by the *LD_PRELOAD* environment variable. You can use this environment variable to add or change functionality when you run a program.

   NOTE:
   For security reasons, the runtime linker unsets *LD_PRELOAD* if the binary has the setuid bit set.

2. the shared library
3. the global list
4. any dependent objects that the shared library references (i.e., any other libraries that the shared library was linked against)

The runtime linker's scoping behavior can be changed in two ways when *dlopen()*'ing a shared library:

- When the program loads a new library, it may instruct the runtime linker to place the library's symbols on the global list by passing the RTLD_GLOBAL flag to the *dlopen()* call. This will make the library's symbols available to any libraries that are subsequently loaded.
- The list of objects that are searched when resolving the symbols within the shared library can be modified. If the RTLD_GROUP flag is passed to *dlopen()*, then only objects that the library directly references will be searched for symbols. If the RTLD_WORLD flag is passed, only the objects on the global list will be searched.

Page updated: August 11, 2025

Filesystems

QNX SDP | 8.0 | System Architecture | Developer | User

The QNX OS provides a rich variety of filesystems. Like most service-providing processes in the OS, these filesystems execute outside the kernel; applications use them by communicating via messages generated by the shared-library implementation of the POSIX API.

Supported filesystems on QNX OS

The following table shows the shared objects and related commands for the filesystems.

The utilities listed under **Initialize with:** create an empty filesystem on a target device or in a file. The ones under **Create with:** create a fully populated filesystem image on a host platform, which you can then write to a target device or a file.

Filesystem	Partition type	Driver	Initialize with:	Create with:	Check with:
Power-Safe filesystem A reliable disk filesystem that can withstand power failures without corruption.	177, 178, or 179	**fs-qnx6.so**	`mkqnx6fs`	`mkqnx6fsimg`	`chkqnx6fs` (on target; not usually necessary)
QNX compressed filesystem A read-only filesystem that supports the compression of files and metadata so that they take up less space.	181	**fs-qcfs.so**	N/A	`mkqfs`	N/A
QNX Trusted Disk A device that provides integrity protection of the underlying disk data in secure boot environments.	185	**fs-qtd.so**	N/A	`mkqfs`	`mkqfs` (on host; use `-y` option to verify)
Image filesystem (IFS) A simple read-only filesystem that presents the set of files built into the OS image.	N/A	`mount_ifs` (secondary IFS) part of the boot image (primary IFS)	N/A	`mkifs`	N/A
DOS filesystem A filesystem that provides transparent access to DOS FAT (FAT12/16/32) disks.	1, 4, 6, 11, 12, or 14	**fs-dos.so**	`mkdosfs`	`mkfatfsimg`	`chkdosfs` (on target)
Linux Ext2 filesystem A filesystem that provides transparent access to Linux disk partitions.	131	**fs-ext2.so**	N/A	N/A	N/A

Filesystem	Partition type	Driver	Initialize with:	Create with:	Check with:
UDF filesystem A read-only filesystem that supports UDF (ECMA-167) and ISO 9660 (including Joliet and RRIP extensions).	N/A	**fs-udf.so**	N/A	N/A	N/A
FFS3 filesystem Filesystem drivers that implement a POSIX-like filesystem on NOR flash memory devices.	N/A	`devf-`*`system`*	`flashctl -f` command to the `devf-`*`system`* driver	`mkefs`	Internal to the `devf-`*`system`* driver, enabled with the `-r` option
NFS filesystem NFS 3 client filesystem.	N/A	**fs-nfs3**	N/A	N/A	N/A
Linux Squash filesystem A compressed, read-only filesystem.	N/A	**fs-squash.so**	N/A	`mksquashfsimg`	N/A

In addition, every QNX OS system provides a simple RAM-based "filesystem" that allows read/write files to be placed under **/dev/shmem**. However, it does not support many POSIX semantics. For more information, see "RAM filesystem."

Filesystem resource managers

These filesystems are *resource managers* as described in this book. Each filesystem adopts a portion of the pathname space (called a *mountpoint*) and provides filesystem services through the standard POSIX API (*open()*, *close()*, *read()*, *write()*, *lseek()*, etc.). Filesystem resource managers take over a mountpoint and manage the directory structure below it. They also check the individual pathname components for permissions and for access authorizations.

This implementation means that:

- Filesystems may be started and stopped dynamically.
- Multiple filesystems may run concurrently.
- Applications are presented with a single unified pathname space and interface, regardless of the configuration and number of underlying filesystems.

Virtual filesystems

A *virtual filesystem* is one in which the files or directories aren't necessarily tied directly to the underlying media, perhaps being manufactured on-demand. The **/proc** filesystem is an example; see "Controlling processes via the **/proc** filesystem" in the Processes chapter of the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

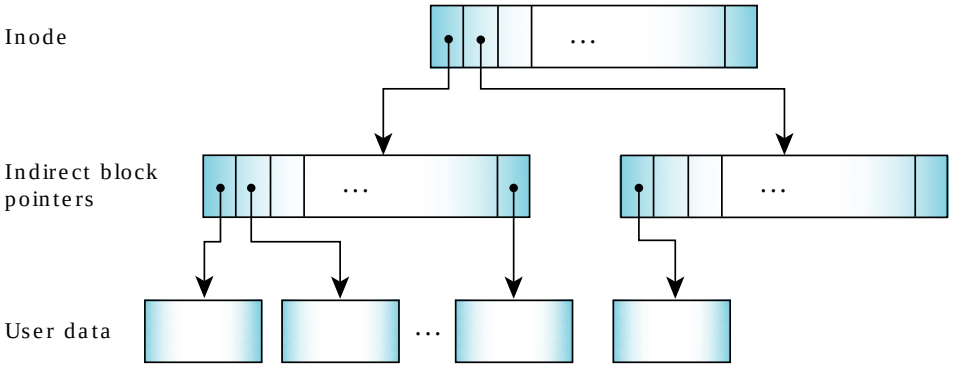
Copy-on-write filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

To address the problems associated with existing disk filesystems, the Power-Safe filesystem never overwrites live data; it does all updates using copy-on-write (COW), assembling a new view of the filesystem in unused blocks on the disk. The new view of the filesystem becomes "live" only when all the updates are safely written on the disk. *Everything* is COW: both metadata and user data are protected.

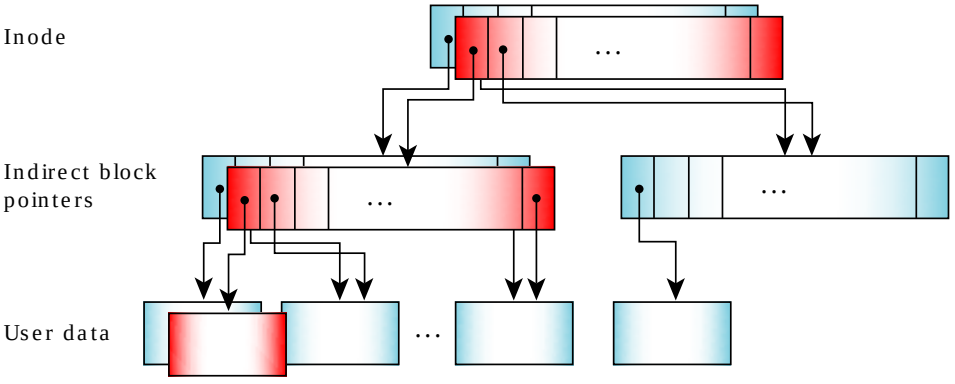
To see how this works, let's consider how the data is stored. A Power-Safe filesystem is divided into logical blocks, the size of which you can specify when you use `mkqnx6fs` to format the filesystem. Each inode includes 16 pointers to blocks. If the file is smaller than 16 blocks, the inode points to the data blocks directly. If the file is any bigger, those 16 blocks become pointers to more blocks, and so on.

The final block pointers to the real data are all in the leaves and are all at the same level. In some other filesystems—such as EXT2—a file always has some direct blocks, some indirect ones, and some double indirect, so you go to different levels to get to different parts of the file. With the Power-Safe filesystem, all the user data for a file is at the same level.



If you change some data, it's written in one or more unused blocks, and the original data remains unchanged. The list of indirect block pointers must be modified to refer to the newly used blocks, but again the filesystem copies the existing block of pointers and modifies the copy. The filesystem then updates the inode—once again by modifying a copy—to refer to the new block of indirect pointers.

When the operation is complete, the original data and the pointers to it remain intact, but there's a new set of blocks, indirect pointers, and inode for the modified data:



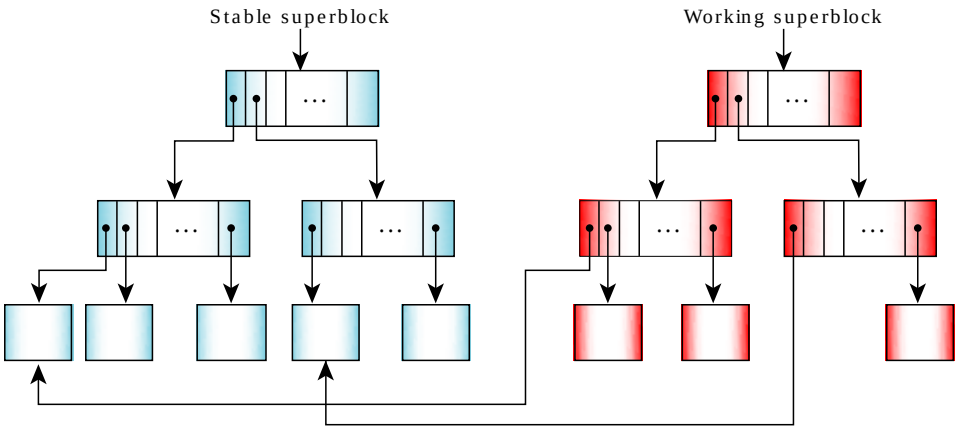
This has several implications for the COW filesystem:

- The bitmap and inodes are treated in the same way as user files.
- Any filesystem block can be relocated, so there aren't any fixed locations.
- The filesystem must be completely self-referential.

A *superblock* is a global root block that contains the inodes for the system bitmap and inodes files. A Power-Safe filesystem maintains *two* superblocks:

- a stable superblock that reflects the original version of all the blocks
- a working superblock that reflects the modified data

The working superblock can include pointers to blocks in the stable superblock. These blocks contain data that hasn't yet been modified. The inodes and bitmap for the working superblock grow from it.



A *snapshot* is a consistent view of the filesystem (simply a committed superblock). To take a snapshot, the filesystem:

1. Locks the filesystem to make sure that it's in a stable state; all client activity is suspended, and there must be no active operations.
2. Writes all the copied blocks to disk. The order isn't important, so it can be optimized.
3. Forces the data to be synchronized to disk, including flushing any hardware track cache.
4. Constructs the superblock, recording the new location of the bitmap and inodes, incrementing its sequence number, and calculating a CRC.
5. Writes the superblock to disk.
6. Switches between the working and committed views. The old versions of the copied blocks are freed and become available for use.

To mount the disk at startup, the filesystem simply reads the superblocks from disk, validates their CRCs, and then chooses the one with the higher sequence number. There's no need to replay a transaction log. The time it takes to mount the filesystem is the time it takes to read a couple of blocks.

NOTE:
If the drive doesn't support synchronizing, **fs-qnx6.so** can't guarantee that the filesystem is power-safe. Before using this filesystem, check to make sure that your device meets the filesystem's requirements. For more information, see “Required properties of the device” in the entry for **fs-qnx6.so** in the *Utilities Reference*.

Page updated: August 11, 2025

RAM “filesystem”

QNX SDP | 8.0 | System Architecture | Developer | User

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

NOTE:
Note that **/dev/shmem** isn't actually a filesystem. It's a window onto the shared memory names that happens to have *some* filesystem-like characteristics.

This RAM filesystem finds the most use in tiny embedded systems where persistent storage across reboots isn't required, yet where a small, fast, *temporary-storage* filesystem with limited features is called for.

The filesystem comes for free with `procnto` and doesn't require any setup. You can simply create files under **/dev/shmem** and grow them to any size (depending on RAM resources).

Although the RAM filesystem itself doesn't support hard or soft links or directories, you can create a link to it by using process-manager links. For example, you could create a link to a RAM-based **/tmp** directory:

```
ln -sP /dev/shmem /tmp
```

This tells `procnto` to create a process manager link to **/dev/shmem** known as “**/tmp**.” Application programs can then open files under **/tmp** as if it were a normal filesystem.

NOTE:
In order to minimize the size of the RAM filesystem code inside the process manager, this filesystem specifically doesn't include “big filesystem” features such as file locking and directory creation.

Page updated: August 11, 2025

DOS filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

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

The structure of the DOS filesystem on disk is old and inefficient, and lacks many desirable features. Its only major virtue is its portability to DOS and Windows environments. You should choose this filesystem only if you need to transport DOS files to other machines that require it. Consider using the Power-Safe filesystem alone if DOS file portability isn't an issue or in conjunction with the DOS filesystem if it is.

If there's no DOS equivalent to a POSIX feature, fs-dos.so, with either return an error or a reasonable default. For example, an attempt to create a *link()* will result in the appropriate *errno* being returned. On the other hand, if there's an attempt to read the POSIX times on a file, **fs-dos.so** will treat any of the *unsupported* times the same as the last write time.

DOS version support

The **fs-dos.so** program supports hard disk partitions from DOS version 2.1 to Windows 98 with long filenames.

DOS text files

DOS terminates each line in a text file with two characters (CR/LF), while POSIX (and most other) systems terminate each line with a single character (LF). Note that **fs-dos.so** makes no attempt to translate text files being read. Most utilities and programs aren't affected by this difference.

Note also that some very old DOS programs may use a **Ctrl Z** (`^Z`) as a file terminator. This character is also passed through without modification.

QNX-to-DOS filename mapping

In DOS, a filename can't contain any of the following characters:

```
/ \ [ ] : * | + = ; , ?
```

An attempt to create a file that contains one of these invalid characters will return an error. DOS (8.3 format) also expects all alphabetical characters to be uppercase, so **fs-dos.so** maps these characters to uppercase when creating a filename on disk. But it maps a filename to lowercase by default when returning a filename to a QNX OS application, so that QNX OS users and programs can always see and type lowercase (via the `sfn=`*sfn_mode* option).

DOS volume labels

DOS uses the concept of a volume label, which is an actual directory entry in the root of the DOS filesystem. To distinguish between the volume label and an actual DOS directory, fs-dos.so reports the volume label according to the way you specify its `vollabel` option. You can choose to:

- Ignore the volume label.
- Display the volume label as a name-special file.
- Display the volume label as a name-special file with an equal sign (`=`) as the first character of the volume name (the default).

Handling filenames

You can specify how you want **fs-dos.so** to handle long filenames (via the `lfn=`*lfn_mode* option):

- Ignore them—display/create only 8.3 filenames.
- Show them—if filenames are longer than 8.3 or if mixed case is used.
- Always create both short and long filenames.

If you use the `ignore` option, you can specify whether or not to silently truncate filename characters beyond the 8.3 limit.

International filenames

The DOS filesystem supports DOS “codepages” (international character sets) for locale filenames. Short 8.3 names are stored using a particular character set (typically the most common extended characters for a locale are encoded in the 8th-bit character range). All common American, Western, and Eastern European codepages (437, 850, 852, 866, 1250, 1251, 1252) are supported. If you produce software that must access a variety of DOS/Windows hard disks, or operate in non-US-English countries, this feature offers important portability—filenames will be created with both a Unicode and locale name and are accessible via either name.

NOTE:

The DOS filesystem supports international text in filenames only. No attempt is made to be aware of data contents, with the sole exception of Windows "shortcut" (**.LNK**) files, which will be parsed and translated into symbolic links if you've specified that option (`lnk=` *lnk_mode*).

DOS-QNX permission mapping

DOS doesn't support all the permission bits specified by POSIX. It has a READ_ONLY bit in place of separate READ and WRITE bits; it doesn't have an EXECUTE bit. When a DOS file is created, the DOS READ_ONLY bit is set if all POSIX WRITE bits are off. When a DOS file is accessed, the POSIX READ bit is always assumed to be set for user, group, and other. Since you can't execute a file that doesn't have EXECUTE permission, **fs-dos.so** has an option (`exe=` *exec_mode*) that lets you specify how to handle the POSIX EXECUTE bit for executables.

File ownership

Although the DOS file structure doesn't support user IDs and group IDs, **fs-dos.so** (by default) doesn't return an error code if an attempt is made to change them. An error isn't returned because a number of utilities attempt to do this and failure would result in unexpected errors. The approach taken is "you can change anything to anything since it isn't written to disk anyway."

The `posix=` options let you set stricter POSIX checks and enable POSIX emulation. For example, in POSIX mode, an error of EINVAL is flagged for attempts to do any of the following:

- Set the user ID or group ID to something other than the default (**root**).
- Remove an `r` (read) permission.
- Set an `s` (set ID on execution) permission.

If you set the `posix` option to `emulate` (the default) or `strict`, you get the following benefits:

- The **.** and **..** directory entries are created in the **root** directory.
- The directory size is calculated.
- The number of links in a directory is calculated, based on its subdirectories.

Page updated: August 11, 2025

Linux Ext2 filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The Linux Ext2 filesystem provides transparent access to Linux disk partitions.

The **fs-ext2.so** implementation supports the standard set of features found in Ext2 versions 0 and 1.

Sparse file support is included in order to be compatible with existing Linux partitions. Other filesystems can only be “stacked” read-only on top of sparse files. There are no such restrictions on normal files.

If an Ext2 filesystem isn't unmounted properly, a filesystem checker is usually responsible for cleaning up the next time the filesystem is mounted. Although the **fs-ext2.so** module is equipped to perform a quick test, it automatically mounts the filesystem as read-only if it detects any significant problems (which should be fixed using a filesystem checker).

Page updated: August 11, 2025

Problems with existing disk filesystems

QNX SDP | 8.0 | System Architecture | Developer | User

Although existing disk filesystems are designed to be robust and reliable, there's still the possibility of losing data, depending on what the filesystem is doing when a catastrophic failure (such as a power failure) occurs.

For example:

- Each sector on a hard disk includes a 4-byte error-correcting code (ECC) that the drive uses to catch hardware errors and so on. If the driver is writing the disk when the power fails, then the heads are removed to prevent them from crashing on the surface, leaving the sector half-written with the new content. The next time you try to read that block—or sector—the inconsistent ECC causes the read to fail, so you lose both the old and new content.

  You can get hard drives that offer atomic sector upgrades and promise you that either all of the old or new data in the sector will be readable, but these drives are rare and expensive.
- Some filesystem operations require updating multiple on-disk data structures. For example, if a program calls *unlink()*, the filesystem has to update a bitmap block, a directory block, and an inode, which means it has to write three separate blocks. If the power fails between writing these blocks, the filesystem will be in an inconsistent state on the disk. Critical filesystem data, such as updates to directories, inodes, extent blocks, and the bitmap are written synchronously to the disk in a carefully chosen order to reduce—but not eliminate—this risk.
- If the root directory, the bitmap, or inode file (all in the first few blocks of the disk) gets corrupted, you wouldn't be able to mount the filesystem at all. You might be able to manually repair the system, but you need to be very familiar with the details of the filesystem structure.

Page updated: August 11, 2025

FFS3 filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The Flash FileSystem version 3 (FFS3) drivers implement a POSIX-like filesystem on NOR flash memory devices. This filesystem is unique to QNX OS.

The drivers are standalone executables that contain both the flash filesystem code and the flash device code. There are versions of FFS3 drivers for different embedded systems hardware as well as PCMCIA memory cards. The naming convention for the drivers is `devf-`*`system`*, where *system* describes the embedded system.

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

- the **boards** and **mtd-flash** directories under *bsp_working_dir***/src/hardware/flash**
- in the QNX OS docs, the `devf-*` entries in the *Utilities Reference*

We provide the libraries and source code needed to build custom flash filesystem drivers for different embedded systems.

Organization

The FFS3 filesystem drivers support one or more logical flash drives. Each logical drive is called a *socket*, which consists of a contiguous and homogeneous region of flash memory. For example, consider a system containing two different types of flash device at different addresses, where one device is used for the boot image and the other for the flash filesystem. In this case, each flash device would appear in a different socket.

Each socket may be divided into one or more partitions. Two types of partitions are supported:

Raw partitions
: A raw partition in the socket is any partition that doesn't contain a flash filesystem. The driver doesn't recognize any filesystem types other than the flash filesystem. A raw partition may contain an image filesystem or some application-specific data.

  The filesystem will make accessible through a raw mountpoint (see below) any partitions on the flash that aren't flash filesystem partitions.

Flash filesystem partitions
: A flash filesystem partition contains the POSIX-like flash filesystem, which uses a proprietary format to store the filesystem data on the flash devices. This format isn't compatible with either the Microsoft Flash File System 2 (FFS2) or the PCMCIA FTL specification.

  The FFS3 filesystem allows files and directories to be freely created and deleted. It recovers space from deleted files using a reclaim mechanism similar to garbage collection.

When you start the flash filesystem driver, by default it will mount any partitions it finds in the socket. There can be only one instance of the driver per flash device, and the driver must be given the full size of the flash chip. You can specify the mountpoint (e.g., **/flash**) using `mkefs` or `flashctl`.

Mountpoint	Description
/dev/fs*X*	Raw mountpoint socket *X*
/dev/fs*X***p***Y*	Raw mountpoint socket *X* partition *Y*
/fs*X***p***Y*	Filesystem mountpoint socket *X* partition *Y*

Features

The FFS3 filesystem supports many advanced features, such as POSIX compatibility, multiple threads, background reclaim, fault recovery, endian-awareness, wear-leveling, and error-handling.

POSIX
: The filesystem supports the standard POSIX functionality (including long filenames, access privileges, random writes, truncation, and symbolic links) with the following exceptions:

  - You can't create hard links.
  - Access times aren't supported (but file modification times and attribute change times are).

  These design compromises allow this filesystem to remain small and simple, yet include most features normally found with block device filesystems.

Storage efficiency

The filesystem uses a fixed-size 32-byte extent header for each piece of user data stored on the filesystem. The filesystem has a 512-byte "append buffer" that's used to coalesce small appends to a file before flushing them out to disk. There's no other caching of user data in the filesystem (read or write). The maximum extent size is currently limited to 4096 bytes of data (plus the 32-byte header) for files which are created at runtime, although `mkefs` can create 16 KB extents.

Wear leveling

Wear leveling is performed on blocks within a given partition. If the flash is split into multiple partitions, then each partition has a smaller pool to level across. Blocks never migrate from one partition to another.

The most common type of reclaim in the filesystem (a foreground reclaim) always does a double-reclaim operation. The first reclaim always takes the block in the partition with the lowest erase count, and swaps it with the spare. Then the block that requires the reclaim is swapped with the spare. This has shown to result in near-perfect wear leveling when foreground reclaims are used.

Background reclaim

The FFS3 filesystem stores files and directories as a linked list of extents, which are marked for deletion as they're deleted or updated. Blocks to be reclaimed are chosen using a simple algorithm that finds the block with the most space to be reclaimed, while keeping level the amount of wear of each individual block. This wear-leveling increases the MTBF (mean time between failures) of the flash devices, thus increasing their longevity.

The background reclaim process is performed when there isn't enough free space. The reclaim process first copies the contents of the reclaim block to an empty spare block, which then replaces the reclaim block. The reclaim block is then erased. Unlike rotating media with a mechanical head, proximity of data isn't a factor with a flash filesystem, so data can be scattered on the media without loss of performance.

Fault recovery

The filesystem has been designed to minimize corruption due to accidental loss-of-power faults. Updates to extent headers and erase block headers are always executed in carefully scheduled sequences. These sequences allow the recovery of the filesystem's integrity in the case of data corruption.

Note that properly designed flash hardware is essential for effective fault-recovery systems. In particular, special reset circuitry must be in place to hold the system in "reset" before power levels drop below critical. Otherwise, spurious or random bus activity can form write/erase commands and corrupt the flash beyond recovery.

Rename operations are guaranteed atomic, even through loss-of-power faults. This means, for example, that if you lost power while giving an image or executable a new name, you would still be able to access the file via its old name upon recovery.

When the FFS3 filesystem driver is started, it scans the state of every extent header on the media (in order to validate its integrity) and takes appropriate action, ranging from a simple block reclamation to the erasure of dangling extent links. This process is merged with the filesystem's normal mount procedure in order to achieve optimal bootstrap timings.

Flash errors

As flash hardware wears out, its write state-machine may find that it can't write or erase a particular bit cell. When this happens, the error status is propagated to the flash driver so it can take proper action (i.e., mark the bad area and try to write/erase in another place).

This error-handling mechanism is transparent. Note that after several flash errors, all writes and erases that fail will eventually render the flash read-only. Fortunately, this situation shouldn't happen before several years of flash operation. Check your flash specification and analyze your application's data flow to flash in order to calculate its potential longevity or MTBF.

Endian awareness

The FFS3 filesystem is endian-aware, making it portable across different platforms. The optimal approach is to use the `mkefs` utility to select the target's endianness.

Utilities

The filesystem supports all the standard POSIX utilities such as `ls`, `mkdir`, `rm`, `ln`, `mv`, and `cp`. There are also some QNX OS utilities for managing the flash:

`flashctl`

Erase, format, and mount flash partitions.

`mkefs`

Create flash filesystem image files.

System calls

The filesystem supports all the standard POSIX I/O functions such as *open()*, *close()*, *read()*, and *write()*. Special functions such as erasing are supported using the non-POSIX *devctl()* function.

Page updated: August 11, 2025

Filesystem classes

QNX SDP | 8.0 | System Architecture | Developer | User

The many available filesystems can be categorized as follows:

Image

A special filesystem that presents the modules in the image and is always present. Note that the `procnto` process automatically provides an image filesystem and a RAM filesystem. For more information, see "Image filesystem" and "RAM "filesystem"".

Block

Traditional filesystems that operate on block devices like hard disks and DVD drives. This includes the Power-Safe filesystem, DOS, and Universal Disk Format filesystems.

Flash

Nonblock-oriented filesystems designed explicitly for the characteristics of flash memory devices. For NOR devices, use the FFS3 filesystem. For NAND devices, contact QNX Technical Support to obtain the ETFS filesystem.

Network

Filesystems that provide network file access to the filesystems on remote host computers (e.g., NFS filesystems).

Virtual

Filesystems in which the files or directories aren't necessarily tied directly to the underlying media, perhaps being manufactured on-demand. This includes the **/proc** filesystem; see "Controlling processes via the **/proc** filesystem" in the Processes chapter of the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

Image filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

Every QNX OS system image provides a simple *read-only* filesystem that presents the set of files built into the OS image.

Since this image may include both executables and data files, this filesystem is sufficient for many embedded systems. If additional filesystems are required, they would be placed as modules within the image where they can be started as needed.

Page updated: August 11, 2025

Filesystem limitations

QNX SDP | 8.0 | System Architecture | Developer | User

POSIX defines the set of services a filesystem must provide. However, not all filesystems are capable of delivering all those services.

Filesystem	Access date	Modification date	Status change date	Filename length[a]	Permissions	Directories	Hard links	Soft links	Decom on rea
Image	No	No	No	255	Yes	No	No	No	No
RAM[b]	Yes	Yes	Yes	255	Yes	No	No	No	No
Power-Safe	Yes	Yes	Yes	510	Yes	Yes	Yes	Yes	No
QCFS	No	Yes	No	255	Yes	Yes	Yes	Yes	Yes
DOS	Yes[c]	Yes	No	8.3[d]	No	Yes	No	No	No
NTFS	Yes	Yes	Yes	755	No	Yes	No	No	Yes
UDF	Yes	Yes	Yes	254	Yes	Yes	No	No	No
FFS3	No	Yes	Yes	255	Yes	Yes	No	Yes	No
NFS	Yes	Yes	Yes	–[e]	Yes[e]	Yes	Yes[e]	Yes[e]	No
Ext2	Yes	Yes	Yes	255	Yes	Yes	Yes	Yes	No
Squash	No[f]	Yes	No[f]	256	Yes	Yes	Yes	Yes	Yes

Page updated: August 11, 2025

[a] Our internal representation for filenames is UTF-8, which has a variable number of bytes per character. Many on-disk formats instead use UCS2, which is a fixed number (2 bytes). Thus a length limit in characters may be 1, 2, or 3 times that number in bytes, as we convert from on-disk to OS representation. The lengths for the Power-Safe and EXT2 filesystems are in bytes; those for UDF and DOS/VFAT are in characters.

[b] The RAM "filesystem" (**/dev/shmem**) isn't really a filesystem; it's a window onto the shared memory names that has *some* filesystem-like characteristics. See "Builtin RAM disk" later in this chapter.

[c] VFAT or FAT32 (Windows 95 or later).

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

[e] Limited by the remote filesystem.

[f] The Squash filesystem contains a single timestamp (the modification time of the original source file) that is advertised as the access, modification, and status change timestamps.

NFS filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The Network File System (NFS) allows a client workstation to perform transparent file access over a network. It allows a client workstation to operate on files that reside on a server across a variety of operating systems. Client file access calls are converted to NFS protocol requests, and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

The Network File System operates in a stateless fashion by using remote procedure calls (RPC) and TCP/IP for its transport. Therefore, to use `fs-nfs3`, you'll also need to run the TCP/IP client for QNX OS.

Any POSIX limitations in the remote server filesystem will be passed through to the client. For example, the length of filenames may vary across servers from different operating systems. NFS version 3 limits filenames to 255 characters; `mountd` (versions 1 and 3) limits pathnames to 1024 characters.

Page updated: August 11, 2025

Filesystems and pathname resolution

QNX SDP | 8.0 | System Architecture | Developer | User

You can seamlessly locate and connect to any service or filesystem that's been registered with the process manager. When a filesystem resource manager registers a mountpoint, the process manager creates an entry in the internal mount table for that mountpoint and its corresponding server ID (i.e., *pid*, *chid* identifiers).

This table effectively joins multiple filesystem directories into what users perceive as a single directory. The process manager handles the mountpoint portion of the pathname; the individual filesystem resource managers take care of the remaining parts of the pathname. Filesystems can be registered (i.e., mounted) in any order.

When a pathname is resolved, the process manager contacts all the filesystem resource managers that can handle some component of that path. The result is a collection of file descriptors that can resolve the pathname.

If the pathname represents a directory, the process manager asks all the filesystems that can resolve the pathname for a listing of files in that directory when *readdir()* is called. If the pathname isn't a directory, then the first filesystem that resolves the pathname is accessed.

For more information on pathname resolution, see the section "Pathname management" in the chapter on the Process Manager in this guide.

Page updated: August 11, 2025

Power-Safe filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The Power-Safe filesystem is a reliable disk filesystem that can withstand power failures without corruption.

This filesystem is supported by the **fs-qnx6.so** shared object. To provide power-safe robustness, the underlying device must have specific properties. For more information, see the **fs-qnx6.so** entry in the *Utilities Reference*.

Expandable filesystem

For the Power-safe filesystem type, you can create an *expandable filesystem*, which is a filesystem with a maximum storage capacity that can exceed its specified size. You can do so using the host-side `mkqnx6fsimg` tool or the target-side `mkqnx6fs` tool; these tools use different units to indicate the filesystem size.

For the `mkqnx6fsimg` tool that builds Power-Safe filesystem images, you must set the `num_sectors` attribute to indicate the number of sectors needed to store the image, and the `max_sectors` attribute to indicate the maximum number of sectors that the filesystem supports. For an explanation of how the maximum capacity is then determined, see the `max_sectors` attribute for the `mkqnx6fsimg` entry in the *Utilities Reference*.

For the `mkqnx6fs` tool that formats Power-Safe filesystems, you must set the `-x` option to request an expandable filesystem, and the `-n` option to indicate the maximum number of logical blocks (i.e., allocatable units) you want to be supported by the filesystem. For details on the required calculation, see the `mkqnx6fs -n` option.

Bitmap and inodes files

In a Power-Safe filesystem, space must be reserved for the bitmap and inodes files.

The bitmap file is a large bitfield that covers as many memory blocks as necessary to represent each block in the filesystem image as a single bit. This bit is 1 if the block is allocated and 0 if not. The bitmap file size is calculated from the maximum number of sectors (`max_sectors`).

The inodes file is a table containing information about particular files in the system. If provided to `mkqnx6fsimg`, the `num_inodes` attribute determines the information nodes (inodes) count. This count is the number of entries in the inodes table. If this attribute isn't provided, `mkqnx6fsimg` determines the inodes count based on other attributes, including `num_sectors`. For more information on this calculation, see the `mkqnx6fsimg num_inodes` attribute. For more details about inodes, see "Links and inodes" in the "Working with Filesystems" chapter of the QNX OS *User's Guide*.

Because these two system files occupy some space in the image, the amount of user-accessible memory is slightly less than the filesystem's maximum capacity.

Access Control Lists (ACLs)

This filesystem type supports ACLs, giving you finer control than `chmod` over the rights of specific users to access specific files. For information about ACL support in QNX OS, refer to the "Working with Access Control Lists (ACLs)" chapter in the *Programmer's Guide*.

Page updated: August 11, 2025

Performance

QNX SDP | 8.0 | System Architecture | Developer | User

The Copy on Write (COW) method has some drawbacks:

- Each change to user data can cause up to a dozen blocks to be copied and modified, because the filesystem never modifies the inode and indirect block pointers in place; it has to copy the blocks to a new location and modify the copies. Thus, write operations are longer.
- When taking a snapshot, the filesystem must force all blocks fully to disk before it commits the superblock.

However:

- There's no constraint on the order in which the blocks (aside from the superblock) can be written.
- The new blocks can be allocated from any free, contiguous space.

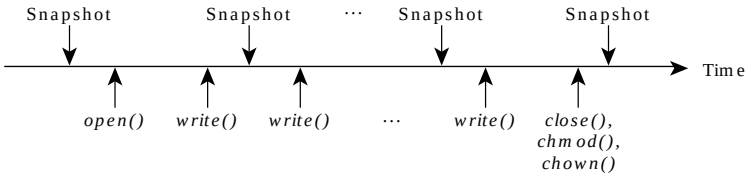
The performance of the filesystem depends on how much buffer cache is available, and on the frequency of the snapshots. Snapshots occur periodically (every 10 seconds, or as specified by the `snapshot` option to **fs-qnx6.so**), and when you call *sync()* for the entire filesystem, or *fsync()* for a single file.

NOTE:
Synchronization is at the filesystem level, not at that of individual files, so *fsync()* is potentially an expensive operation; the Power-Safe filesystem ignores the O_SYNC flag.

You can also turn snapshots off if you're doing some long operation, and the intermediate states aren't useful to you. For example, suppose you're copying a very large file into a Power-Safe filesystem. The `cp` utility is really just a sequence of basic operations:

- an `open(O_CREAT|O_TRUNC)` to make the file
- a bunch of *write()* operations to copy the data
- a *close()*, *chmod()*, and *chown()* to copy the metadata

If the file is big enough so that copying it spans snapshots, you have on-disk views that include the file not existing, the file existing at a variety of sizes, and finally the complete file copied and its IDs and permissions set:



Each snapshot is a valid point-in-time view of the filesystem (i.e., if you've copied 50 MB, the size is 50 MB, and all data up to 50 MB is also correctly copied and available). If there's a power failure, the filesystem is restored to the most recent snapshot. But the filesystem has no concept that the sequence of *open()*, *write()*, and *close()* operations is really one higher-level operation, `cp`. If you want the higher-level semantics, disable the snapshots around the `cp`, and then the middle snapshots won't happen, and if a power failure occurs, the file will be either complete or not there at all.

For information about using this filesystem, see "Power-Safe filesystem" in the Working with Filesystems chapter of the QNX OS *User's Guide*.

Page updated: August 11, 2025

Filesystems as shared libraries

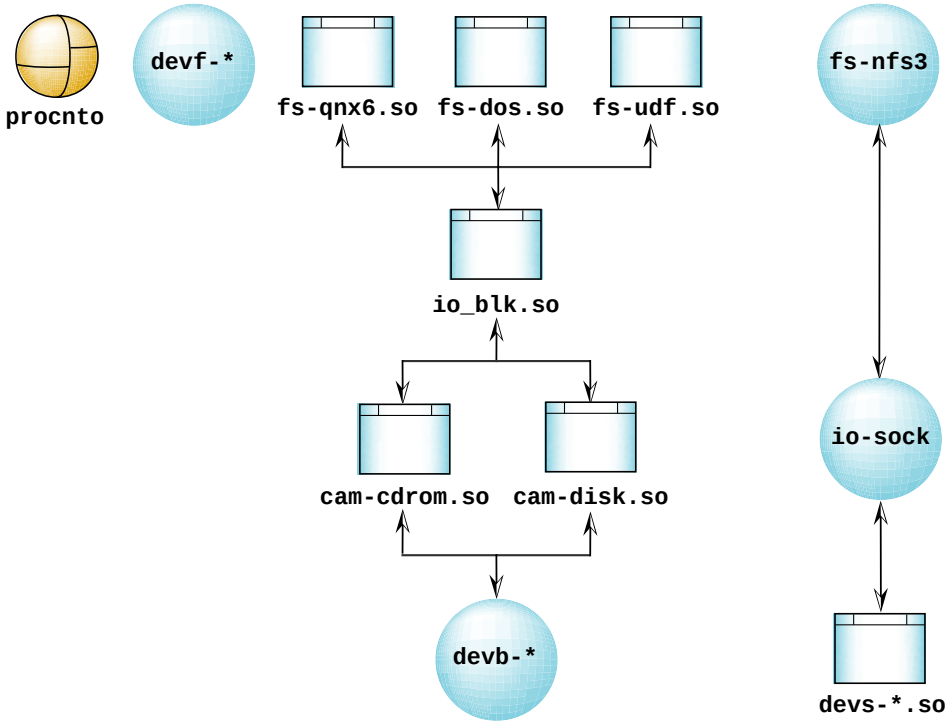
QNX SDP | 8.0 | System Architecture | Developer | User

Since it's common to run many filesystems under the QNX OS, they have been designed as a family of drivers and shared libraries to maximize code reuse. This means the cost of adding an additional filesystem is typically smaller than might otherwise be expected.

Once an initial filesystem is running, the incremental memory cost for additional filesystems is minimal, since only the code to implement the new filesystem protocol would be added to the system.

The various filesystems are layered as follows:

Figure 1QNX OS filesystem layering.



As shown in this diagram, the filesystems and `io-blk` are implemented as shared libraries (essentially passive blocks of code resident in memory), while the `devb-*` driver is the executing process that calls into the libraries. In operation, the driver process starts first and invokes the block-level shared library (`io-blk.so`). The filesystem shared libraries may be dynamically loaded later to provide filesystem interfaces and services.

A "filesystem" shared library implements a filesystem protocol or "personality" on a set of blocks on a physical disk device. The filesystems aren't built into the OS kernel; rather, they're dynamic entities that can be loaded or unloaded on demand.

For example, a removable storage device (removable cartridge disk, etc.) may be inserted at any time, with any of a number of filesystems stored on it. While the hardware the driver interfaces to is unlikely to change dynamically, the on-disk data structure could vary widely. The dynamic nature of the filesystem copes with this very naturally.

Page updated: August 11, 2025

Linux Squash filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The Linux Squash filesystem (`squashfs`) is a compressed read-only filesystem that allows you to compress data, inodes, and directories.

The **fs-squash.so** shared object provides access to Squash filesystems. QNX OS support for `squashfs` allows experienced Linux developers to use a familiar method for compressing data, which is useful in size-constrained embedded systems.

Page updated: August 11, 2025

UDF filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The Universal Disk Format (UDF) filesystem provides access to recordable media, such as CD, CD-R, CD-RW, and DVD. It's used for DVD video, but can also be used for backups to CD, and so on. For more information, see http://osta.org/specs/index.htm.

The UDF filesystem is supported by the fs-udf.so shared object.

NOTE:
In our implementation, UDF filesystems are read-only.

Page updated: August 11, 2025

Encryption

QNX SDP | 8.0 | System Architecture | Developer | User

The Power-Safe filesystem can provide *data at rest* encryption to all or part of its content.

NOTE:

- When you use an encrypted filesystem, read/write throughput rates and performance are lower and CPU usage is higher than when you use a nonencrypted filesystem.
- You can't mount a partition that contains encrypted files unless **/dev/random** is running. If `random` is using an entropy file that is stored on an encrypted filesystem, make sure that you start `random` with `-S`.

You might need to encrypt different parts of the filesystem with different keys, and you might not need to encrypt some parts, so the Power-Safe filesystem lets you create multiple *encryption domains*, which you can lock or unlock as needed.

A domain can contain any number of files or directories, but a file or directory can belong to at most one domain. If you assign a domain to a directory, all files subsequently created in that directory are encrypted and inherit that domain. Assigning a domain to a directory doesn't introduce encryption to any files or directories that are already in it.

You can simply assign a domain to a directory or an empty file; the filesystem treats them in the same way because they don't have any content to encrypt. If you want to encrypt a file that isn't empty, you must make it *migrate* to a domain because the filesystem needs to decrypt the file, assign the new domain to it, and then encrypt the file again.

Files that are assigned to a domain are encrypted on disk, and the files' contents are available only when the associated domain is unlocked. When a domain is unlocked, all its files and directories—regardless of their locations in the volume—are unlocked as well, and are therefore accessible (as per basic file permissions). When a domain is locked, any access to file content belonging to that domain is denied. This design offers *data at rest* encryption, meaning that file content is always encrypted before being written to disk.

NOTE:
Locking and unlocking operations apply to an entire domain, not to specific files or directories.

Domain 0 (FS_CRYPTO_UNASSIGNED_DOMAIN) is always unlocked, and its contents are unencrypted; you can design your system to use any other domains. Valid domain numbers are 0–119.

To use encryption, you must set the `crypto=` option for **fs-qnx6.so**. You can then use `fsencrypt` to manage the encryption. The `chkqnx6fs` utility automatically identifies the encryption format and verifies the integrity of the encryption data. It's also possible for you to use `fsencrypt` to enable encryption after you've formatted the volume and added content, but you must have set the `crypto=` option when you started **fs-qnx6.so**.

Key types

Key type	Description
File key	Private and randomly generated at the time the file is created (if the file is assigned to a domain). Holds some information about the file to ensure integrity between a file and its key. Used to encrypt file data, and is encrypted by a domain key. Keys are managed by the filesystem and are hidden from the user.
Domain key	Private and randomly generated at the time the domain is created. Used to encrypt all the file keys that belong to its domain, and is encrypted by a domain master key. Keys are managed by the filesystem and are hidden from the user.
Master key	Optionally public, as it is supplied and managed by a third party (not the filesystem). Used to encrypt the domain key, required on domain creation and subsequent unlock requests.

Encryption types

The Power-Safe filesystem supports the following types of encryption:

Domain-encryption type	Constant	Description	Key length
0	FS_CRYPTO_TYPE_NONE	No encryption	–
1	FS_CRYPTO_TYPE_XTS	AES-256, in XTS mode. The two keys are randomly generated.	512 bits
2	FS_CRYPTO_TYPE_CBC	AES-256, in CBC mode	256 bits
3–99	–	Reserved for future use	–

Interface usage

To manage encryption from the command line, use `fsencrypt`; its `-c` option lets you specify the command to run. From your code, use the **fscrypto** library. You need to include both the **<fs_crypto_api.h>** and **<sys/fs_crypto.h>** header files. Many of the APIs return EOK on success and have a reply argument that provides more information.

API	`fsencrypt` command	Description
fs_crypto_check()	`check`	Determine if the underlying filesystem supports encryption
fs_crypto_domain_add()	`create`	Create the given domain/type if it doesn't already exist. You need to provide a 64-bit encryption key. From a program, you can create a locked or unlocked domain; `fsencrypt` always creates an unlocked domain.
fs_crypto_domain_add_flags()	`create`	Create the given domain/type if it doesn't already exist, with the given locked state and flags for new file and domain keys. You need to provide a 64-bit encryption key.
fs_crypto_domain_hard_lock()	`lock`	Lock a domain, preventing access to the original contents of any file belonging to it. With this function, the lock behavior is controlled by the FS_CRYPTO_HARD_LOCK_ACTION_ENFORCE flag. If this flag is *not* set, access is limited to clients that appear on a whitelist. If this flag is set, no clients have access.
fs_crypto_domain_key_change()	`change-key`	Change the master domain key used to encrypt the domain key
fs_crypto_domain_key_check()	`check-key`	Determine if a given domain key is valid
fs_crypto_domain_key_size()	–	Return the size of keys needed for filesystem encryption
fs_crypto_domain_lock()	`lock`	Lock a domain, preventing access to the original contents of any file belonging to it. You must be in the group that owns the filesystem's mountpoint to lock a domain.
fs_crypto_domain_query()	`query`, `query-all`	Get status information for a domain

API	`fsencrypt` command	Description
fs_crypto_domain_remove()	`destroy`	Destroy a domain. You must be in the group that owns the filesystem's mountpoint. It isn't possible to retrieve any files in the domain after it's been destroyed.
fs_crypto_domain_unlock()	`unlock`	Unlock a domain, given the appropriate key data
fs_crypto_domain_whitelist_configure()	–	Perform whitelist configuration for a domain whitelist
fs_crypto_domain_whitelist_ctrl()	–	Perform a control action for a domain whitelist
fs_crypto_domain_whitelist_ctrl_access_grant()	–	Grant a client access to a domain
fs_crypto_domain_whitelist_ctrl_access_revoke()	–	Revoke a client's access to a domain
fs_crypto_domain_whitelist_get_flags()	–	Get the flags for a domain whitelist
fs_crypto_domain_whitelist_set_flags()	–	Set the flags for a domain whitelist. Whitelists are effective only when you're using a hard lock and you've not set the FS_CRYPTO_HARD_LOCK_ACTION_ENFORCE flag; for more details, see *fs_crypto_domain_hard_lock()* above.
fs_crypto_enable(), *fs_crypto_enable_option()*	`enable`	Enable encryption support on a volume that wasn't set up for it at formatting time
fs_crypto_file_get_domain()	`get`	Return the domain of a file or directory, if assigned
fs_crypto_file_set_domain()	`set`	Assign a given domain to the path (a regular file or a directory). Regular files must have a length of zero. The domain replaces any domain previously assigned to the path.
fs_crypto_key_gen()	`-K` or `-k` option	Generate an encryption key from a password
fs_crypto_set_logging()	`-l` and `-v` options	Set the logging destination and verbosity

The library also includes some functions that you can use to move existing files and directories into an encryption domain. To do this, you unlock the source and destination domains, tag the files and directories that you want to move, and then start the migration, which the filesystem does in the background. These functions include:

API	`fsencrypt` commands	Description
fs_crypto_migrate_control()	`migrate-start`, `migrate-stop`, `migrate-delay`, `migrate-units`	Control encryption migration within the filesystem
fs_crypto_migrate_path()	`migrate-path`	Mark an entire directory for migration into an encryption domain
fs_crypto_migrate_status()	`migrate-status`	Get the status of migration in the filesystem
fs_crypto_migrate_tag()	`migrate-tag`, `tag`	Mark a file for migration into an encryption domain

Examples

Here are some examples of the way you can use `fsencrypt` to manage filesystem encryption:

- Determine if encryption is supported or enabled:

```
$ fsencrypt -vc check -p /
ENCRYPTION_CHECK(Path:'/') FAILED: (18) - 'No support'
```

- Enable encryption on an existing filesystem:

```
$ fsencrypt -vc enable -p /
ENCRYPTION_CHECK(Path:'/') SUCCESS
$ fsencrypt -vc check -p /
ENCRYPTION_CHECK(Path:'/') NOTICE: Encryption is SUPPORTED)
```

NOTE:
This change is irreversible and forces two consecutive disk transactions to rewrite some data in the superblocks.

- Determine if a file is encrypted. Files with the domain number of 0 aren't encrypted. A nonzero value means the file is assigned to a domain. Access to the file's original contents is determined by the status of the domain. The example below shows that the named file is assigned to domain 10:

```
$ fsencrypt -vcget -p /accounts/1000/secure/testfile
GET_DOMAIN(Path:'/accounts/1000/secure/testfile') = 10 SUCCESS
```

- Determine if a domain is locked. If a domain is locked, reading the files within that domain yields encrypted data; unlocked domains yield the original file contents. “Unused” domains are ones that haven't yet been created. In the example below, domain 11 hasn't yet been created, and domain 10 is currently unlocked:

```
$ fsencrypt -vc query -p/ -d11
QUERY_DOMAIN(Path:'/', Domain:11) NOTICE: Domain is UNUSED
$ fsencrypt -vc query -p/ -d10
QUERY_DOMAIN(Path:'/', Domain:10) NOTICE: Domain is UNLOCKED
```

Page updated: August 11, 2025

Filesystem events

QNX SDP | 8.0 | System Architecture | Developer | User

Keeping up with a very “live” filesystem that changes often and quickly is a challenge for the programs that work with it. To help with this challenge, the QNX OS `io-blk.so` library and, thus, all `devb-*` drivers, support *inotify_*()* functions.

`inotify_event`
: *Structure that describes a watched filesystem event*

inotify_add_watch()
: *Add or update a watch for filesystem events associated with a path*

inotify_init()
: *Initialize the inode notify system*

inotify_rm_watch()
: *Remove the watch associated with the given watch descriptor*

inotify_qnx_ext()
: *Enable QNX OS-specific extensions to the inotify interface*

The **sys/inotify.h** header file includes everything needed to read and process events from the event manager. For more information, see the *inotify_*()* functions in the *C Library Reference*.

While the QNX OS inotify implementation follows the same semantics as the Linux implementation, the behavior differs from Linux due to the kernel and filesystem models employed. For an overview of inotify, see *inotify_add_watch()* from Linux Standard Base Core Specification 4.1. Additionally, see *inotify_init()* and *inotify_rm_watch()*.

Page updated: August 11, 2025

io-blk

QNX SDP | 8.0 | System Architecture | Developer | User

Most of the filesystem shared libraries ride on top of the Block I/O module.

The `io-blk.so` module also acts as a resource manager and exports a block-special file for each physical device. For a system with two hard disks the default files would be:

/dev/hd0
: First hard disk.

/dev/hd1
: Second hard disk.

These files represent each raw disk and may be accessed using all the normal POSIX file primitives (*open()*, *close()*, *read()*, *write()*, *lseek()*, etc.). Although the `io-blk` module can support a 64-bit offset on seek, the driver interface is 32-bit, allowing access to 2-terabyte disks.

Page updated: August 11, 2025

QNX compressed filesystem

QNX SDP | 8.0 | System Architecture | Developer | User

The QNX compressed filesystem (QCFS) provides a read-only filesystem that supports the compression of files and metadata so that they take up less space.

The **fs-qcfs.so** shared object supports the QCFS.

It includes the following features:

- directory indexes
- metadata compression
- sparse file support
- tail-end packing (fragments)
- hard link support
- a single or a double period entry (`.` or `..`) in *readdir()*
- real inode numbers
- 32-bit user or group IDs
- file creation time
- ACL support

For information on generating a QCFS image, see the `mkqfs` utility. For information about ACL support in QNX OS, refer to the "Working with Access Control Lists (ACLs)" chapter in the *Programmer's Guide*.

Supported compression algorithms

QCFS supports the following compression algorithms:

`lz4hc`
: LZ4 high compression algorithm, which provides the best decompression speed at the cost of a lesser compression ratio.

`zstd`
: Zstandard compression algorithm, which provides a good compression ratio and decryption speed.

Page updated: August 11, 2025

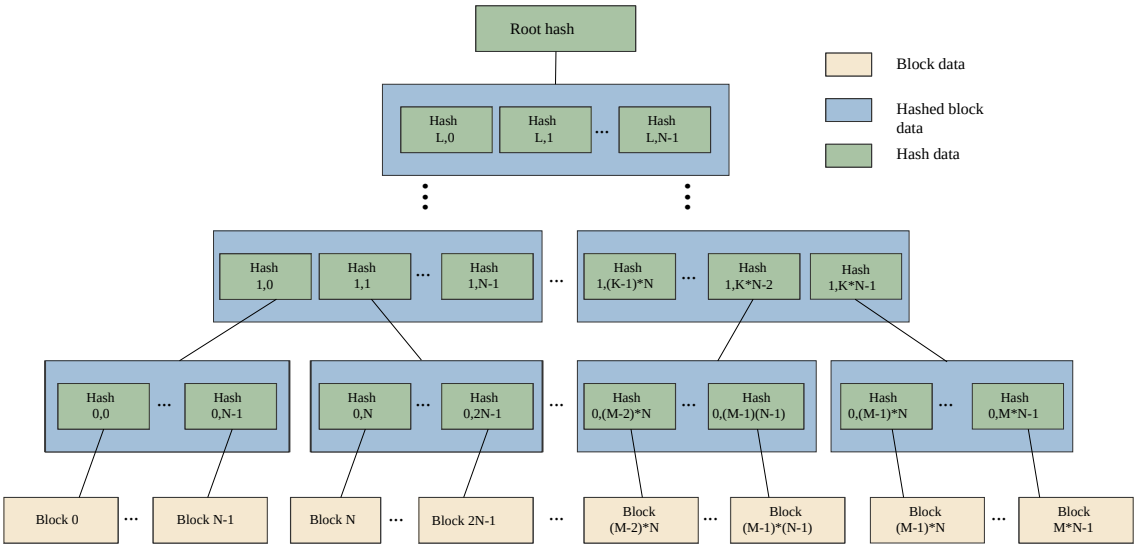
QNX Trusted Disk

QNX SDP | 8.0 | System Architecture | Developer | User

QNX Trusted Disk (QTD) devices provide integrity protection of the underlying disk data in secure boot environments. They can extend the secure boot chain up to the core operating system filesystem that stores the critical binaries and configuration files.

The QTD protection mechanism is based on a Merkle tree and is supported by the fs-qtd.so shared object.

When building a QTD image, a metadata hash tree is constructed from the blocks of the source filesystem image.



QTD disk devices are read-only. The QTD driver sits between the raw block device and the upper filesystem layer that is supported (for example, a Power-Safe filesystem supported by fs-qnx6.so). On read access, it uses the hash tree metadata to verify the integrity of the data. If the verification fails, an error is returned instead.

The QTD metadata is signed and verified using a key pair. Verifying the signature ensures that the root hash of the tree is valid and hasn't been tampered with. It is the root of the trusted verification mechanism.

The size of the QTD image depends on the chosen block size as well as the chosen digest algorithm. You can use the `mkqfs` utility to generate statistics that describe how much additional space the metadata consumes.

Using QTD as a protected container (app model)

You can use QTD as a package container solution by mounting files that are themselves QTD images. For an example, see the fs-qtd.so entry in the *Utilities Reference*.

Secure hash algorithms

Use the following table to choose the best performing algorithm based on the desired security strength of the digest function.

	Digest security	
Architecture	**128 bits**	**256 bits**
64-bit	sha512-256 blake2b256 blake3	sha512 blake2b512

Signing keys

Refer to the `mkqfs` utility for the supported key types and signing algorithms.

Crypto engines

QTD uses the cryptographic algorithms available via the QNX cryptography library (**qcrypto**). It can use any **qcrypto** plugin that supports the chosen digest algorithm and signature type. For more information, see "QNX Cryptography Library".

Page updated: August 11, 2025

Glossary

QNX SDP | 8.0 | Utilities Reference | Utilities

application ID

A number that identifies all processes that are part of an application. Like process group IDs, the application ID value is the same as the process ID of the first process in the application. A new application is created by spawning with the POSIX_SPAWN_NEWAPP or SPAWN_NEWAPP flag. A process created without one of those inherits the application ID of its parent. A process needs the PROCMGR_AID_CHILD_NEWAPP ability in order to set those flags.

The *SignalKill()* kernel call accepts a SIG_APPID flag ORed into the signal number parameter. This tells it to send the signal to all the processes with an application ID that matches the *pid* argument. The DCMD_PROC_INFO *devctl()* returns the application ID in a structure field.

asymmetric multiprocessing (AMP)

A multiprocessing system where a separate OS, or a separate instantiation of the same OS, runs on each CPU.

atomic

Of or relating to atoms. :-)

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

attributes structure

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

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

bank-switched

A term indicating that a certain memory component (usually the device holding an *image*) isn't entirely addressable by the processor. In this case, a hardware component manifests a small portion (or "window") of the device onto the processor's address bus. Special commands have to be issued to the hardware to move the window to different locations in the device. See also *linearly mapped*.

base layer calls

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

BIOS/ROM Monitor extension signature

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

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

block-integral

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

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

bootable

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

bootfile

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

bound multiprocessing (BMP)

A more restrictive case of symmetric multiprocessing (SMP) in which individual threads are locked, or bound, to specific CPUs. For more information, see SMP below.

budget

In *sporadic* scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for `mkifs` specifying the contents and other details of an *image*, or for `mkefs` specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or "cooked" mode. In this mode, the character device library performs line-editing operations on each received character. Only when a line is "completely entered"—typically when a carriage return (CR) is received—will the line of data be made available to application processes. Contrast *raw mode*.

card information structure (CIS)

A data block that maintains information about flash configuration. The CIS description includes the types of memory devices in the regions, the physical geometry of these devices, and the partitions located on the flash.

channel

A kernel object used with message passing.

In QNX OS, message passing is directed towards a *connection* (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using *ChannelCreate()*), and then receives messages from that channel (using *MsgReceive()*). Another thread that wishes to send a message to the first thread must make a connection to that channel by "attaching" to the channel (using *ConnectAttach()*) and then sending data (using *MsgSend()*).

chid

An abbreviation for *channel ID*.

cluster

A group of associated processors. Some clusters are always defined, including one representing all the processors in the system, and a set of clusters where each one represents a different processor. So on a system with *N* processors, there are always (at least) *N+1* clusters.

Further clusters may be defined by the startup program.

coid

An abbreviation for *connection ID*.

combine message

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

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

Common Internet File System (CIFS)

also known as Server Message Block (SMB)

A protocol that allows a client workstation to perform transparent file access over a network to a Windows server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

connect message

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

connection

A kernel object used with message passing.

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

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

context

Information retained between invocations of functionality.

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

cooked mode

See *canonical mode*.

core dump

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

critical section

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

deadlock

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

design safe state (DSS)

Generally speaking, a well-defined state that the kernel transitions to when it encounters a situation that it cannot handle. For QNX OS, moving to DSS includes outputting certain important kernel variables and other information, such as a shutdown report, by invoking the *display_char()* kernel callout, and then invoking the *reboot()* kernel callout. The intent of DSS is to stop the system as soon as behavior requirements can no longer be guaranteed. Calling *reboot()* ensures that the entire system enters a safe state.

device driver

A process that allows the OS and application programs to use the underlying hardware in a generic way (e.g., a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for the QNX OS are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn't affect any other part of the OS—drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won't cause the entire OS to shut down.

Domain Name System (DNS)

An Internet protocol used to convert ASCII domain names into IP addresses.

dynamic bootfile

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

dynamic linking

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

edge-sensitive

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

edited mode

See *canonical mode*.

End of Interrupt (EOI)

A command that the OS sends to the programmable interrupt controller (PIC) after masking the current interrupt level and before scheduling any Interrupt Service Threads (ISTs) for this interrupt. This tells the PIC to reset the processor's In Service Register. See also *PIC* and *IST*.

erasable programmable read-only memory (EPROM)

A memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g., 12 V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state.

Changing a bit from a 0 state into a 1 state can be accomplished only by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically 10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast with *flash* and *RAM*.

event

A scheme used to notify a thread that a particular condition has occurred. QNX has several mechanisms for notifying a thread about the occurrence of a condition, including pulses, signals, semaphores, updates to memory, thread creation, unblocking of kernel calls, and unblocking of ISTs.

file descriptor (FD)

A handle that a client process uses to access a resource such as a file, socket, pipe, etc. In a QNX system, a file descriptor is also a connection ID that the client obtained by connecting to a server by a call such as *open()* or *socket()*. The client can then use the file descriptor in subsequent calls to operate on the same resource.

first in, first out (FIFO)

A scheduling policy whereby a thread is able to consume CPU at its priority level without bounds. Also refer to *round robin* and *sporadic*.

flash memory

A memory technology similar in characteristics to *EPROM* memory, with the exception that erasing is performed electrically instead of via ultraviolet light, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64 KB at a time) instead of the entire device. Contrast *EPROM* and *RAM*.

garbage collection

Also known as space reclamation, the process whereby a filesystem manager recovers the space occupied by deleted files and directories.

handle

A pointer that the resource manager base library binds to the pathname registered via *resmgr_attach()*. This handle is typically used to associate some kind of per-device information. Note that if you use the *iofunc_*() POSIX layer calls*, you must use a particular *type* of handle—in this case called an *attributes structure*.

hardware interrupt

See *interrupt*.

high availability (HA)

In telecommunications and other industries, HA describes a system's ability to remain up and running without interruption for extended periods of time.

I/O message

A message that relies on an existing binding between the client and the resource manager. For example, an _IO_READ message requires that the client has established an association (or *context*) with the resource manager by issuing an *open()* and getting back a file descriptor. Also refer to *connect message*, *context*, *combine message*, and *message*.

I/O privileges

Particular rights, that, if enabled for a given thread, allow the thread to perform I/O instructions (such as the x86 assembler `in` and `out` instructions). By default, I/O privileges are disabled because a program with them enabled can disrupt the system. To enable I/O privileges, the process must have the PROCMGR_AID_IO ability enabled (see *procmgr_ability()*), and the thread must call *ThreadCtl()*.

image

In the context of embedded QNX OS systems, an image can mean either a structure that contains files (i.e., an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim FFS3-compatible filesystem (i.e., a flash filesystem image).

inherit mask

A bitmask that specifies which processors a thread's children can run on. Contrast with *runmask*.

initial program loader (IPL)

The software component that either takes control at the processor's reset vector (e.g., location 0xFFFFFFF0 on the x86), or is a BIOS extension. This component is responsible for putting a machine into a usable state such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also *BIOS extension signature* and *startup code*.

interprocess communication (IPC)

The ability for two processes (or threads) to communicate. The QNX OS offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), and signals.

interrupt

An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it to do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.

interrupt handler

See *interrupt service routine*.

interrupt latency

The amount of time elapsed between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service thread (IST) returning from its blocking function. Also designated as "T_{il}".

interrupt request (IRQ)

A hardware request line asserted by a peripheral to indicate that it requires servicing by software. The IRQ is handled by the *PIC*, which then interrupts the processor, usually causing the processor to execute an *Interrupt Service Thread (IST)*.

interrupt service routine (ISR)

A routine responsible for servicing hardware (e.g., reading and/or writing some device ports), for updating some data structures shared between the ISR and the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.

NOTE:
QNX OS does not use ISRs anymore; see ISTs.

interrupt service thread (IST)

A thread that is responsible for servicing an interrupt.

The thread attaches to the interrupt via *InterruptAttachThread()* or *InterruptAttachEvent()*, and then blocks waiting for the QNX OS to unblock it after an interrupt has happened.

kernel

See *microkernel*.

level-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast *edge-sensitive*.

linearly mapped

A term indicating that a certain memory component is entirely addressable by the processor. Contrast *bank-switched*.

message

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

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various "changes of state" that affect when, and for how long, they may run.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mount structure

An optional, well-defined data structure (of type `iofunc_mount_t`) within an *iofunc_*()* structure, which contains information used on a per-mountpoint basis (generally used only for filesystem resource managers). See also *attributes structure* and *OCB*.

mountpoint

The location in the pathname space where a resource manager has "registered" itself. For example, the serial port resource manager registers mountpoints for each serial device (**/dev/ser1**, **/dev/ser2**, etc.), and a CD-ROM filesystem may register a single mountpoint of **/cdrom**.

multicore system

A system that has one physical processor with multiple execution units (cores) interconnected over a chip-level bus.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (*pthread_mutex_lock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a *critical section*). See also *critical section*.

Network File System (NFS)

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

nonbootable

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

nonmaskable interrupt (NMI)

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

Open Control Block (OCB)

also known as Open Context Block

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

pathname prefix

See *mountpoint*.

pathname space mapping

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

persistent

When applied to storage media, the ability for the media to retain information across a power-cycle. For example, a hard disk is a persistent storage medium, whereas a ramdisk is not, because the data is lost when power is lost.

pid

An abbreviation for *process ID* (e.g., as an argument in a function call). See also process ID.

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface—the "X" alludes to "UNIX", on which the interface is based.

POSIX layer calls

Convenient set of library calls for writing resource managers. The POSIX layer calls can handle even more of the common-case messages and functions than the *base layer calls*. These calls are identified by the *iofunc_*()* prefix. In order to use these (and we strongly recommend that you do), you must also use the well-defined POSIX-layer *attributes* (`iofunc_attr_t`), *OCB* (`iofunc_ocb_t`), and (optionally) *mount* (`iofunc_mount_t`) structures.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been "preempted" by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY on the same CPU, the lower-priority thread is immediately preempted by the higher-priority thread.

prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent *priority inversion*.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU *on behalf of* the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it's working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one *thread* running in it—this thread is then called the first thread.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.

process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of 1.

processor affinity

A user-specified binding of a thread to a set of processors, done by means of a *runmask*.

programmable interrupt controller (PIC)

A hardware component that handles IRQs. See also *edge-sensitive*, *level-sensitive*, and *IST*.

pseudo-TTY (pty)

A character-based device that has a controller end and a worker end. Data written to the controller end shows up on the worker end and vice versa. These devices are typically used to interface between two programs, one that expects a character device and another that wishes to use that device (e.g., the shell and the `sshd` process).

pulses

In addition to the synchronous Send/Receive/Reply services, QNX OS also supports fixed-size, nonblocking messages known as pulses. These carry a small payload (eight bytes of data plus a single byte code). A pulse is also one form of *event* that can be generated by a timer or attached to an interrupt. See *MsgDeliverEvent()* for more information.

random access memory (RAM)

A memory technology characterized by the ability to read and write any location in the device without limitation. Contrast *flash* and *EPROM*.

Rate Monotonic Analysis (RMA)

A set of methods used to specify, analyze, and predict the timing behavior of realtime systems.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data—you don't want any translations of the raw binary stream between the device and the application. Contrast *canonical mode*.

replenishment

In *sporadic* scheduling, the period of time during which a thread is allowed to consume its execution *budget*.

reset vector

The address at which the processor begins executing instructions after the processor's reset line has been activated. On the x86, for example, this is the address 0xFFFFFFF0.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX OS resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g., serial ports, parallel ports, network cards, disk drives) or virtual (e.g., **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers*. But unlike device drivers, QNX OS resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also *device driver*.

round robin

A scheduling policy whereby a thread is given a certain period of time to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also *FIFO*, and *sporadic*.

runmask

A bitmask that indicates which processors a thread can run on. It must match a *cluster* defined for the system.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast *static linking*.

scheduling latency

The amount of time elapsed between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as "T_{sl}".

scoid

An abbreviation for *server connection ID*.

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

socket

A virtual endpoint for communication. For example, in TCP/IP, a socket is a combination of an IP address and a port number that uniquely identifies a single network process. Another example is the Unix domain socket, which exchanges data between processes that execute on the same host operating system.

software interrupt

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

sporadic

A scheduling policy whereby a thread's priority can oscillate dynamically between a "foreground" or normal priority and a "background" or low priority. A thread is given an execution *budget* of time to be consumed within a certain *replenishment* period. See also *FIFO*, and *round robin*.

startup code

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

static bootfile

An image created at one time and then transmitted whenever a node boots. Contrast *dynamic bootfile*.

static linking

The process whereby you combine your modules with the modules from the library to form a single executable that's entirely self-contained. The word "static" implies that it's not going to change—*all* the required modules are already combined into one.

symmetric multiprocessing (SMP)

A multiprocessor system where a single instantiation of an OS manages all CPUs simultaneously. In this system, threads can float to any CPU; that is, they may run on any of them, or they can be bound to specific CPUs.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under the QNX OS. A thread is a flow of execution; it exists within the context of a *process*.

tid

An abbreviation for *thread ID*.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an *event*, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a *round-robin* scheduled thread. This period of time is small (on the order of tens of milliseconds); the actual value shouldn't be relied upon by any program (it's considered bad design).

translation look-aside buffer (TLB)

A cache of page table entries. To maintain performance, the processor caches frequently used portions of the external memory page tables in the TLB.

Page updated: August 11, 2025

High Availability

QNX SDP | 8.0 | System Architecture | Developer | User

The term *High Availability* (HA) is commonly used in telecommunications and other industries to describe a system's ability to remain up and running without interruption for extended periods of time.

The celebrated "five nines" availability metric refers to the percentage of uptime a system can sustain in a year—99.999% uptime amounts to about five minutes of downtime per year.

Obviously, an effective HA solution involves various hardware and software components that combine to form a stable, working system. Assuming reliable hardware components with sufficient redundancy, how can an OS best remain stable and responsive when a particular component or application program fails? And in cases where redundant hardware may not be an option (e.g., consumer appliances), how can the OS itself support HA?

Page updated: August 11, 2025

HAM API

QNX SDP | 8.0 | System Architecture | Developer | User

The basic mechanism to talk to the HAM is to use its API. This API is implemented as a library that you can link against. The library is thread-safe and cancellation-safe.

To control exactly what/how you're monitoring, the HAM API provides a collection of functions, including:

Function	Description
ham_action_execute()	Add an execute action to a condition.
ham_action_fail_execute()	Add to an action an execute action that will be executed if the corresponding action fails.
ham_action_fail_log()	Insert a log message into the activity log.
ham_action_fail_notify_pulse()	Add to an action a notify pulse action that will be executed if the corresponding action fails.
ham_action_fail_notify_signal()	Add to an action a notify signal action that will be executed if the corresponding action fails.
ham_action_fail_remove()	Remove an action-fail item from an action.
ham_action_fail_waitfor()	Add to an action a waitfor action that will be executed if the corresponding action fails.
ham_action_handle()	Get a handle to an action in a condition in an entity.
ham_action_handle_free()	Free a previously obtained handle to an action in a condition in an entity.
ham_action_heartbeat_healthy()	Reset a heartbeat's state to healthy.
ham_action_log()	Insert a log message into the activity log.
ham_action_notify_pulse()	Add a notify-pulse action to a condition.
ham_action_notify_signal()	Add a notify-signal action to a condition.
ham_action_remove()	Remove an action from a condition.
ham_action_restart()	Add a restart action to a condition.
ham_action_waitfor()	Add a waitfor action to a condition.
ham_attach()	Attach an entity.
ham_attach_self()	Attach an application as a self-attached entity.
ham_condition()	Set up a condition to be triggered when a certain event occurs.
ham_condition_handle()	Get a handle to a condition in an entity.
ham_condition_handle_free()	Free a previously obtained handle to a condition in an entity.
ham_condition_raise()	Attach a condition associated with a condition raise condition that's triggered by an entity raising a condition.
ham_condition_remove()	Remove a condition from an entity.
ham_condition_state()	Attach a condition associated with a state transition condition that's triggered by an entity reporting a state change.
ham_connect()	Connect to a HAM.
ham_detach()	Detach an entity from a HAM.
ham_detach_name()	Detach an entity from a HAM, using an entity name.

Function	Description
ham_detach_self()	Detach a self-attached entity from a HAM.
ham_disconnect()	Disconnect from a HAM.
ham_entity()	Create entity placeholder objects in a HAM.
ham_entity_condition_raise()	Raise a condition.
ham_entity_condition_state()	Notify the HAM of a state transition.
ham_entity_handle()	Get a handle to an entity.
ham_entity_handle_free()	Free a previously obtained handle to an entity.
ham_heartbeat()	Send a heartbeat to a HAM.
ham_heartbeat_control()	Check the current state of, suspend, or resume heartbeat handling in a HAM.
ham_stop()	Stop a HAM.
ham_verbose()	Modify the verbosity of a HAM.

Page updated: August 11, 2025

Custom hardware support

QNX SDP | 8.0 | System Architecture | Developer | User

While many operating systems provide HA support in a hardware-specific way (e.g., via PCI Hot Plug), QNX OS isn't tied to PCI. Your particular HA system may be built on a custom chassis, in which case an OS that offers a PCI-based HA “solution” may not address your needs at all.

Page updated: August 11, 2025

High Availability Manager

QNX SDP | 8.0 | System Architecture | Developer | User

The High Availability Manager (HAM) provides a mechanism for monitoring processes and services on your system.

The goal is to provide a resilient manager (or “smart watchdog”) that can perform multistage recovery whenever system services or processes fail, no longer respond, or are detected to be in a state where they cease to provide acceptable levels of service.

The HA framework, including the HAM, uses a simple publish/subscribe mechanism to communicate interesting system events between interested components in the system.

The HAM acts as a conduit through which the rest of the system can both obtain and deliver information regarding the state of the system as a whole. The HAM can monitor specific processes and can control the behavior of the system when specific components fail and need to be recovered. The HAM also allows external detectors to detect and report interesting events to the system, and can associate actions with the occurrence of those events.

In many HA systems, each single point of failure (SPOF) must be identified and dealt with carefully. Since the HAM maintains information about the health of the system and also provides the basic recovery framework, the HAM itself must never become a SPOF.

Page updated: August 11, 2025

The HAM as a “filesystem”

QNX SDP | 8.0 | System Architecture | Developer | User

Effectively, the HAM's internal state is like a hierarchical filesystem, where entities are like directories, conditions associated with those entities are like subdirectories, and actions inside those conditions are like leaf nodes of this tree structure.

The HAM also presents this state as a read-only filesystem under **/proc/ham**. As a result, arbitrary processes can also view the current state (e.g., you can do `ls /proc/ham`).

The **/proc/ham** filesystem presents a lot of information about the current state of the system's entities. It also provides useful statistics on heartbeats, restarts, and deaths, giving you a snapshot in time of the system's various entities, conditions, and actions.

Page updated: August 11, 2025

HAM hierarchy

QNX SDP | 8.0 | System Architecture | Developer | User

The High Availability Manager consists of three main components.

- Entities
- Conditions
- Actions

Page updated: August 11, 2025

HA-specific modules

QNX SDP | 8.0 | System Architecture | Developer | User

Apart from its inherently robust architecture, QNX OS also provides an HA Manager—a “smart watchdog” that can perform multistage recovery whenever system services or processes fail.

Page updated: August 11, 2025

Inherent HA

QNX SDP | 8.0 | System Architecture | Developer | User

A true microkernel that provides full memory protection is inherently the most stable OS architecture.

Very little code is running in kernel mode that could cause the kernel itself to fail. And individual processes, whether applications or OS services, can be started and stopped dynamically, without jeopardizing system uptime.

QNX OS inherently provides several key features that are well-suited for HA systems:

- System stability through full memory protection for all OS and user processes.
- Dynamic loading and unloading of system components (device drivers, filesystem managers, etc.).
- Separation of all software components for simpler development and maintenance.

While any claims regarding “five nines” availability on the part of an OS must be viewed only in the context of the entire hardware/software HA system, one can always ask whether an OS truly has the appropriate underlying architecture capable of supporting HA.

Page updated: August 11, 2025

Multistage recovery

QNX SDP | 8.0 | System Architecture | Developer | User

The HAM can perform a multistage recovery, executing several actions in a certain order. This technique is useful whenever strict dependencies exist between various actions in a sequence. In most cases, recovery requires more than a single restart mechanism in order to properly restore the system's state to what it was before a failure.

For example, suppose you've started `fs-nfs3` (the NFS filesystem) and then mounted a few directories from multiple sources. You can instruct the HAM to restart `fs-nfs3` upon failure, and also to remount the appropriate directories as required after restarting the NFS process.

As another example, suppose `io-sock` (the network I/O manager) were to die. We can tell the HAM to restart it and also to load the appropriate network drivers (and maybe a few more services that essentially depend on network services in order to function).

Page updated: August 11, 2025

Publishing autonomously detected conditions

QNX SDP | 8.0 | System Architecture | Developer | User

Entities or other components in the system can inform the HAM about conditions (events) that they deem interesting, and the HAM in turn can deliver these conditions (events) to other components in the system that have expressed interest in (subscribed to) them.

This publishing feature allows arbitrary components that are capable of detecting error conditions (or potentially erroneous conditions) to report these to the HAM, which in turn can notify other components to start corrective and/or preventive action.

There are currently two different ways of publishing information to the HAM; both of these are designed to be general enough to permit clients to build more complex information exchange mechanisms:

- publishing state transitions
- publishing other conditions.

Page updated: August 11, 2025

An OS for HA

QNX SDP | 8.0 | System Architecture | Developer | User

If you had to design an HA-capable OS from the ground up, would you start with a single executable environment? In this simple, high-performance design, all OS components, device drivers, applications, the works, would all run without memory protection in kernel mode.

On second thought, maybe such an OS wouldn't be suited for HA, simply because if a single software component were to fail, the entire system would crash. And if you wanted to add a software component or otherwise modify the HA system, you'd have to take the system out of service to do so. In other words, the *conventional realtime executive* architecture wasn't built with HA in mind.

Suppose, then, that you base your HA-enabled OS on a separation of kernel space and user space, so that all applications would run in user mode and enjoy memory protection. You'd even be able to upgrade an application without incurring any downtime.

So far so good, but what would happen if a device driver, filesystem manager, or other essential OS component were to crash? Or what if you needed to add a new driver to a live system? You'd have to rebuild and restart the kernel. Based on such a *monolithic kernel* architecture, your HA system wouldn't be as available as it should be.

Page updated: August 11, 2025

The HAM and the Guardian

QNX SDP | 8.0 | System Architecture | Developer | User

As a self-monitoring manager, the HAM is resilient to internal failures. If, for whatever reason, the HAM itself is stopped abnormally, it can immediately and completely reconstruct its own state. A mirror process called the *Guardian* perpetually stands ready and waiting to take over the HAM's role. Since all state information is maintained in shared memory, the Guardian can assume the exact same state that the original HAM was in before the failure.

But what happens if the Guardian terminates abnormally? The Guardian (now the new HAM) creates a new Guardian for itself *before taking the place of the original HAM*. Practically speaking, therefore, one can't exist without the other.

Since the HAM/Guardian pair monitor each other, the failure of either one can be completely recovered from. The only way to stop the HAM is to explicitly instruct it to terminate the Guardian and then to terminate itself.

Page updated: August 11, 2025

Subscribing to autonomously published conditions

QNX SDP | 8.0 | System Architecture | Developer | User

To express their interest in events published by other components, subscribers can use the *ham_condition_state()* and *ham_condition_raise()* API calls.

These are similar to the *ham_condition()* API call (e.g., they return a handle to a condition), but they allow the subscriber customize which of several possible published conditions they're interested in.

Trigger based on state transition

When an entity publishes a state transition, a *state transition condition* is raised for that entity, based on the two states involved in the transition (the *from* state and the *to* state). Subscribers indicate which states they're interested in by specifying values for the *fromstate* and *tostate* parameters in the *ham_condition_state()* API call.

Trigger based on specific published condition

To express interest in conditions raised by entities, subscribers can use the API call *ham_condition_raise()*, indicating as parameters to the call what sort of conditions they're interested in.

For more information, see the API reference documentation in the High Availability Framework *Developer's Guide*.

Page updated: August 11, 2025

The Philosophy of the QNX OS

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

Page updated: August 11, 2025

Interprocess communication

QNX SDP | 8.0 | System Architecture | Developer | User

When several threads run concurrently, as in typical realtime multitasking environments, the OS must provide mechanisms to allow them to communicate with each other. Interprocess communication (IPC) is the key to designing an application as a set of cooperating processes in which each process handles one well-defined part of the whole.

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

Page updated: August 11, 2025

Microkernel architecture

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

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

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

Figure 1Conventional executives offer no memory protection.

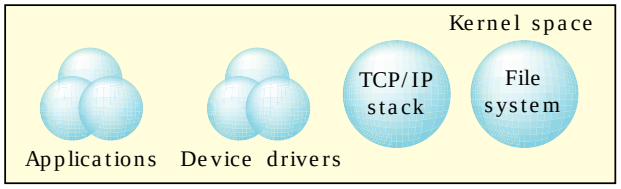


Figure 2In a monolithic OS, system processes have no protection.

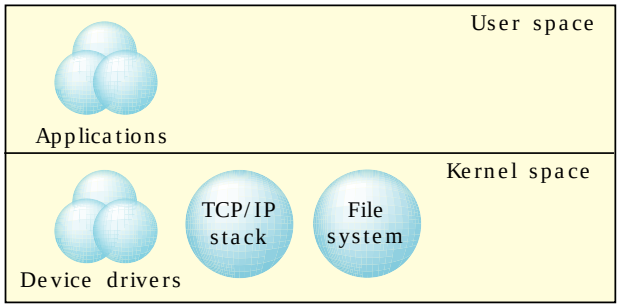
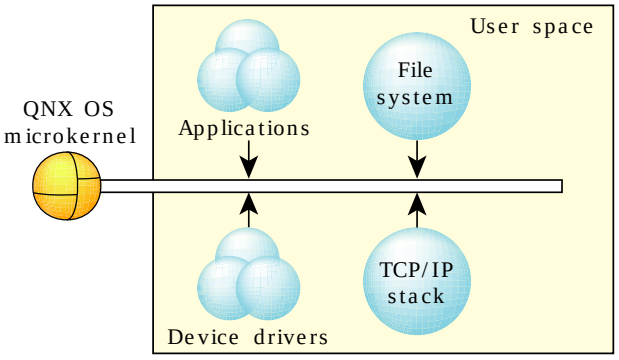


Figure 3A microkernel provides complete memory protection.



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

Page updated: August 11, 2025

QNX OS as a message-passing operating system

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

Page updated: August 11, 2025

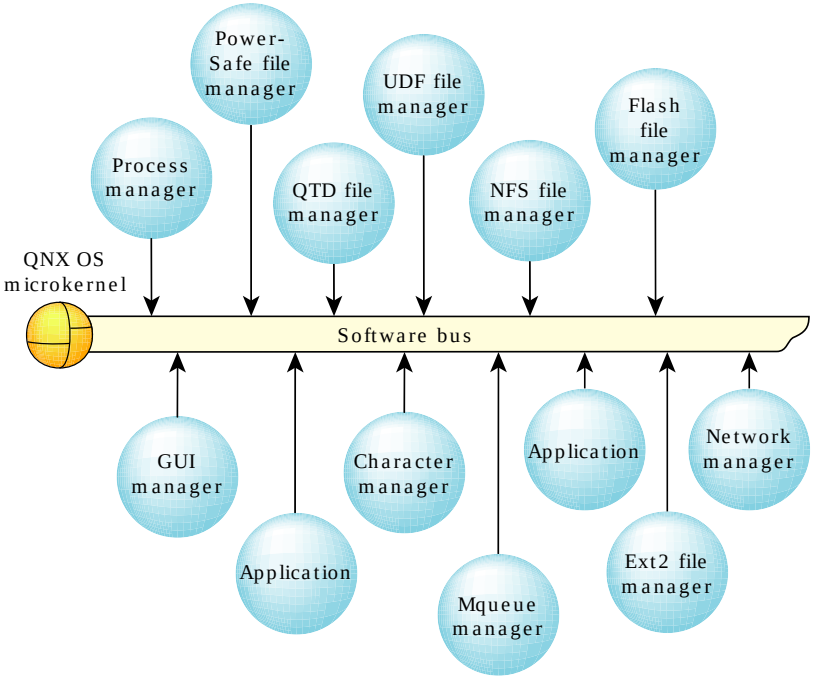
The OS as a team of processes

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Figure 1The QNX OS architecture.



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

Page updated: August 11, 2025

Product scaling

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Some of the advantages to this scalable approach include:

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

Page updated: August 11, 2025

System processes

QNX SDP | 8.0 | System Architecture | Developer | User

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

A richly configured system could include the following:

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

System processes vs user-written processes

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

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

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

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

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

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

Device drivers

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

Page updated: August 11, 2025

A true kernel

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

Page updated: August 11, 2025

Why QNX OS for embedded systems?

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

- microkernel architecture
- message-based interprocess communication

Page updated: August 11, 2025

Why POSIX for embedded systems?

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

Multiple OS sources

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

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

Portability of development staff

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

Development environment

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

Page updated: August 11, 2025

An embeddable POSIX OS?

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

Page updated: August 11, 2025

Interprocess Communication (IPC)

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

QNX OS offers at least the following forms of IPC:

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

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

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

Page updated: August 11, 2025

Channels and connections

QNX SDP | 8.0 | System Architecture | Developer | User

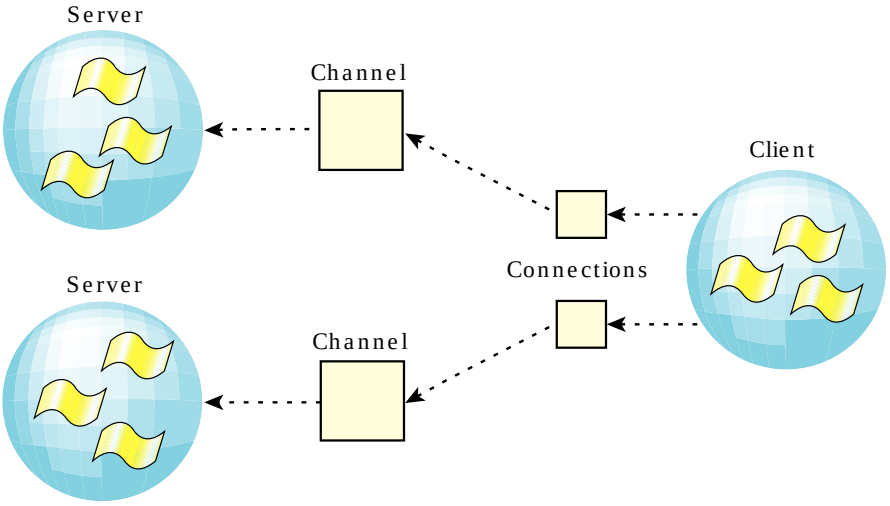
In the QNX OS, message passing is directed towards channels and connections, rather than targeted directly from thread to thread. A thread that wishes to receive messages first creates a channel; another thread that wishes to send a message to that thread must first make a connection by "attaching" to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

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

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

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

Figure 1Connections map elegantly into file descriptors.



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

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

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

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

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

NOTE:

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

The channel has several lists of messages associated with it:

Receive

A LIFO queue of threads waiting for messages.

Send

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

Reply

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

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

Page updated: August 11, 2025

Creating a shared memory object

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

POSIX shared memory is implemented in the QNX OS via the process manager (`procnto`). The above calls are implemented as messages to `procnto` (see the "Process Manager" chapter).

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

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

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

Shared memory objects that are populated with *shm_ctl()* are implicitly locked.

Page updated: August 11, 2025

Events

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

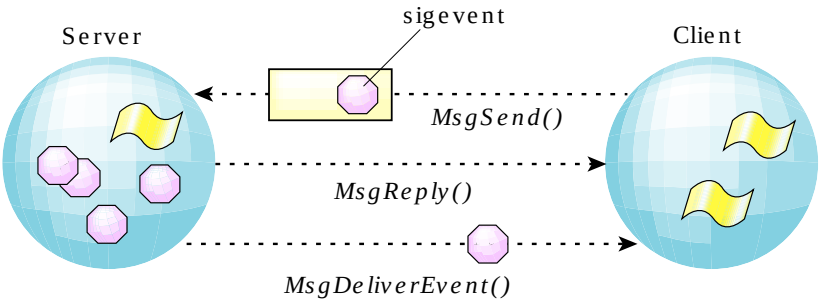
- a *MsgDeliverEvent()* kernel call invoked by a thread
- the expiry of a timer
- A POSIX message queue transitioning from empty to non-empty.

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

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

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

Figure 1The client sends a `sigevent` to the server.



For details about the `sigevent` structure, see its entry in the *C Library Reference*.

In a deeply embedded system that consists entirely of trusted programs, no safeguards are needed on events, but this isn't true in systems that are more open. A problem with `sigevents` is that a server can deliver any event to a client, even if the client has never expressed an interest in receiving events.

The client can register events to make them more secure, as follows:

1. The client sets up the `sigevent` to indicate how it wants to be notified.
2. If the client wants to allow the server to update the value associated with the event, it sets the SIGEV_FLAG_UPDATEABLE flag in the event.
3. The client registers an event by calling *MsgRegisterEvent()*. This function registers the given `sigevent` with the kernel, which stores the event internally and provides a handle for the event.
4. The client gives the handle instead of the actual event to the server.
5. If the client set the SIGEV_FLAG_UPDATEABLE flag on the event, then the server is allowed to update the value included in the registered event.
6. When the server calls *MsgDeliverEvent()* with a handle, the kernel looks up the handle. If the handle exists and the client has allowed the server to deliver it, the kernel delivers the corresponding registered event to the client.

7. When the client no longer needs the secure event, it can call *MsgUnregisterEvent()* to remove it and unregister the handle.

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

You can detect when processes use unregistered events using `secpolgenerate` (via the file **/dev/secpolgenerate/errors**) or `secpolmonitor` (when `-u` is specified).

Page updated: August 11, 2025

I/O notification

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

Page updated: August 11, 2025

Message-passing API

QNX SDP | 8.0 | System Architecture | Developer | User

The message-passing API consists of the functions listed here.

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

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

Page updated: August 11, 2025

Message copying

QNX SDP | 8.0 | System Architecture | Developer | User

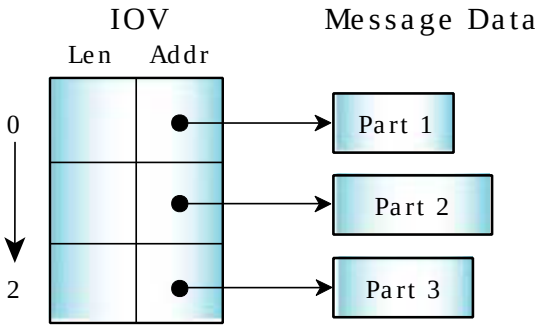
Since our messaging services copy a message directly from the address space of one thread to another without intermediate buffering, the message-delivery performance approaches the memory bandwidth of the underlying hardware.

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

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

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

Figure 1A multipart transfer.

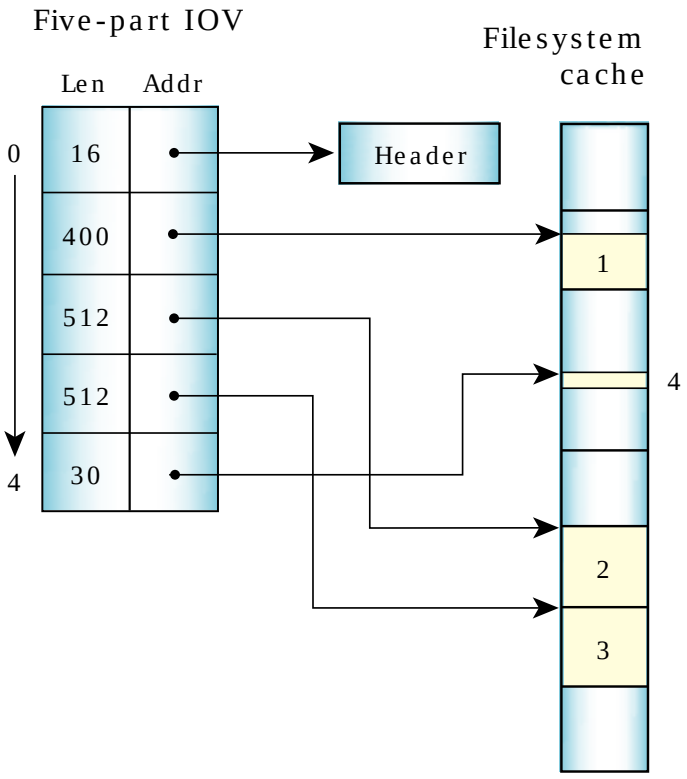


Each IOV can have a maximum of 524288 parts. The sum of the sizes of the parts must not exceed SSIZE_MAX.

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

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

Figure 2Scatter/gather of a read of 1454 bytes.



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

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

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

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

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

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

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

NOTE:

Since many messages passed are small, copying such messages twice though an intermediate buffer is faster than manipulating page tables to do a direct process-to-process copy, so the kernel does this as an optimization for short messages.

POSIX message queues

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

NOTE:
To use POSIX message queues, `mqueue`, the message queue server, must be running (see the "Resource Managers" chapter in this book) .

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

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

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

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

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

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

In QNX OS, all message queues created appear in the filename space under the directory **/dev/mqueue**.

For example:

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

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

```
ls /dev/mqueue
```

Page updated: August 11, 2025

Pipes and FIFOs

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Pipes

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

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

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

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

```
ls | more
```

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

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

FIFOs

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

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

Page updated: August 11, 2025

Priority inheritance and messages

QNX SDP | 8.0 | System Architecture | Developer | User

QNX OS uses message-driven priority inheritance to avoid priority-inversion problems.

A server process receives messages and pulses in priority order. The threads that receive messages within the server inherit the priorities of the sending threads and pulses. As a result, the relative priorities of the threads requesting work of the server are preserved, and the server work will be executed at the appropriate priority.

For example, suppose the system includes the following:

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

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

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

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

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

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

The server thread's priority does *not* revert to its previous setting when the thread replies to a client. The new priority remains in effect until one of the following occurs:

- The priority is boosted when another message is sent from a higher-priority thread or a higher-priority pulse is sent.
- The priority is lowered when a message is received from a lower-priority thread or a lower-priority pulse is received.
- The priority is explicitly set via a function call such as *pthread_setschedparam()* or *pthread_setschedprio()*.

NOTE:
You can turn off priority inheritance by specifying the _NTO_CHF_FIXED_PRIORITY flag when you call *ChannelCreate()*.

Page updated: August 11, 2025

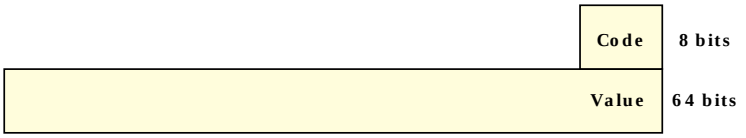
Pulses

QNX SDP | 8.0 | System Architecture | Developer | User

In addition to the synchronous Send/Receive/Reply services, the OS also supports fixed-size, nonblocking messages. These are referred to as *pulses* and carry a small payload.

Pulses pack a relatively small payload—eight bits (one byte) of code and 64 bits (eight bytes) of data. Pulses are often used to allow servers to notify clients without blocking on them; for example, to notify that data has become available on a hardware device.

Figure 1Pulses pack a small payload.



Page updated: August 11, 2025

Robust implementations with Send/Receive/Reply

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

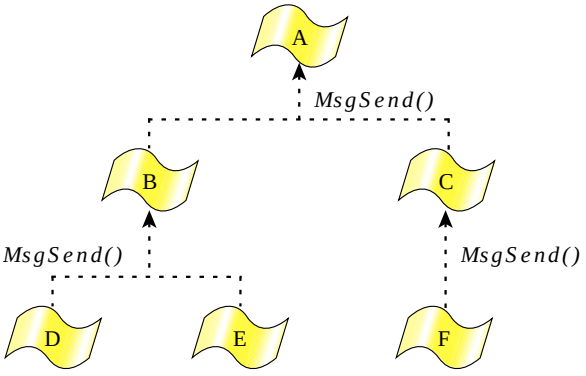
Avoiding deadlock situations is another difficult problem when constructing applications from various combinations of queued IPC, shared memory, and miscellaneous synchronization primitives. For example, suppose thread A doesn't release mutex 1 until thread B releases mutex 2. Unfortunately, if thread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

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

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

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

Figure 1Threads should always send up to higher-level threads.



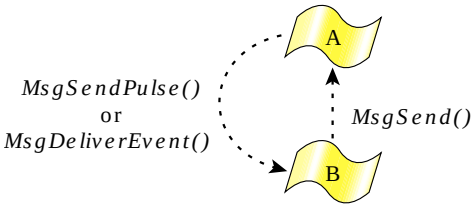
Here the threads at any given level in the hierarchy never send to each other, but send only upwards instead.

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

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

The QNX OS provides a very flexible architecture with the *MsgDeliverEvent()* kernel call to deliver nonblocking events. All of the common asynchronous services can be implemented with this. For example, the server-side of the *poll()* call is an API that an application can use to allow a thread to wait for an I/O event to complete on a set of file descriptors. In addition to an asynchronous notification mechanism being needed as a "back channel" for notifications from higher-level threads to lower-level threads, we can also build a reliable notification system for timers, hardware interrupts, and other event sources around this.

Figure 2A higher-level thread can "send" a pulse event.



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

Page updated: August 11, 2025

Server boost

QNX SDP | 8.0 | System Architecture | Developer | User

Priority inheritance relies on deciding which server thread needs to be boosted when a high-priority client sends a message. If there are server threads available to receive a message (i.e., RECEIVE-blocked), then the kernel boosts the thread that receives the message. But if no server threads are available and, thus, the client is left SEND-blocked, it's not clear which thread should be boosted.

Without a RECEIVE-blocked server thread, when a client sends a message or pulse, the kernel boosts the priority of one server thread that's associated with the channel. Priority inversion can still happen—this is a known limitation—but it won't prevent any threads from finishing their work and doing a receive.

NOTE:
This situation is considered to be a symptom of a poorly designed or configured server, and the kernel's response is an attempt to work around it. For a multithreaded server, when a new client request is made, the server should always have at least one RECEIVE-blocked thread. Following this design prevents priority inversion for client processes.

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

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

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

How a thread becomes associated with a channel

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

Page updated: August 11, 2025

Shared memory

QNX SDP | 8.0 | System Architecture | Developer | User

Shared memory offers the highest bandwidth IPC available.

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

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

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

Page updated: August 11, 2025

Shared memory with message passing

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

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

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

Page updated: August 11, 2025

Summary of signals

QNX SDP | 8.0 | System Architecture | Developer | User

This table describes what each signal means and the default action that QNX OS performs to handle it. Here, *default action* means what happens if no handler for this specific signal is defined by the application writer.

Signal	Description	Default action
SIGABRT	Abnormal termination, issued by functions such as *abort()*	Kill the process and write a dump file
SIGALRM	Alarm clock, issued by functions such as *alarm()*	Kill the process
SIGBUS	Bus error, or a memory parity error (a QNX OS-specific interpretation). If a second fault occurs while your process is in a signal handler for this fault, the process is terminated.	Kill the process and write a dump file
SIGCHLD or SIGCLD	A child process terminated or stopped	Ignore the signal, but still let the process's children become zombies
SIGCONT	Continue the process. You can't block this signal.	Make the process continue if it's STOPPED; otherwise ignore the signal
SIGDEADLK	A mutex deadlock occurred. QNX doesn't generate this signal, it's user-generated (i.e., generated by a user call to a function such as *kill*, *raise*, *pthread_kill*, etc.). SIGDEADLK and SIGEMT refer to the same signal. Some utilities (e.g., `gdb`, `ksh`, `slay`, and `kill`) know about SIGEMT, but not SIGDEADLK.	Kill the process and write a dump file
SIGDOOM	Terminate the process due to insufficient resources (e.g., it's out of memory). This is similar to SIGKILL but you can ignore it.	Kill the process
SIGEMT	EMT instruction (emulation trap) SIGDEADLK and SIGEMT refer to the same signal. Some utilities (e.g., `gdb`, `ksh`, `slay`, and `kill`) know about SIGEMT but not SIGDEADLK.	Kill the process and write a dump file
SIGFPE	Floating point exception	Kill the process and write a dump file. This signal is delivered for a given floating-point exception only if it is unmasked. By default, all floating-point exceptions are masked. To unmask them, you must call *feenableexcept()*.
SIGHUP	Hangup; the session leader died, or the controlling terminal closed	Kill the process
SIGILL[a]	Illegal hardware instruction. If a second fault occurs while your thread is in a signal handler for this fault, the process is terminated.	Kill the process and write a dump file
SIGINT	Interrupt; typically generated when you press **CtrlC** or **CtrlBreak** (you can change this with `stty`)	Kill the process
SIGIO	Asynchronous I/O	Ignore the signal
SIGIOT	I/O trap; a synonym for SIGABRT	Kill the process
SIGKILL	Kill. You can't block, ignore, or catch this signal.	Kill the process[b]

Signal	Description	Default action
SIGPIPE	Write on pipe with no reader	Kill the process
SIGPOLL	System V name for SIGIO	Ignore the signal
SIGPROF	Profiling timer expired. POSIX has marked this signal as obsolescent; QNX OS doesn't support profiling timers or send this signal.	Kill the process
SIGPWR	Power failure	Ignore the signal
SIGQUIT	Quit; typically generated when you press **Ctrl** (you can change this with `stty`)	Kill the process and write a dump file
SIGSEGV	Segmentation violation; an invalid memory reference was detected. If a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.	Kill the process and write a dump file
SIGSTOP	Stop the process. You can't block, ignore, or catch this signal.	Stop the process[b]
SIGSYS	Bad argument to system call	Kill the process and write a dump file
SIGTERM	Termination signal	Kill the process
SIGTRAP	Trace trap	Kill the process and write a dump file
SIGTSTP	Stop signal from tty; typically generated when you press **CtrlZ** (you can change this with `stty`)	Stop the process
SIGTTIN	Background read attempted from control terminal	Stop the process
SIGTTOU	Background write attempted to control terminal	Stop the process
SIGURG	Urgent condition on I/O channel	Ignore the signal
SIGUSR1	User-defined signal 1	Kill the process
SIGUSR2	User-defined signal 2	Kill the process
SIGVTALRM	Virtual timer expired. POSIX has marked this signal as obsolescent; QNX OS doesn't support virtual timers or send this signal.	Kill the process
SIGWINCH	The size of the terminal window changed	Ignore the signal
SIGXCPU	Soft CPU time limit exceeded (see the RLIMIT_CPU resource for *setrlimit()*)	Kill the process and write a dump file
SIGXFSZ	File size exceeded (see the RLIMIT_FSIZE resource for *prlimit()*) This happens if the hard limit on disk quotas has been exceeded on a Power-Safe (QNX6) filesystem (see Managing disk quotas) in the QNX OS User's Guide).	Kill the process and write a dump file

Page updated: August 11, 2025

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

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

[b] For signals that can't be ignored or caught, meaning application writers can't tell the OS to throw them away and can't define handlers for them or wait for them, they are always handled by the OS and, hence, the default action is always performed.

Signals

QNX SDP | 8.0 | System Architecture | Developer | User

The OS supports the 32 standard POSIX signals (as in UNIX) as well as the POSIX realtime signals, both numbered from a kernel-implemented set of 64 signals with uniform functionality.

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

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

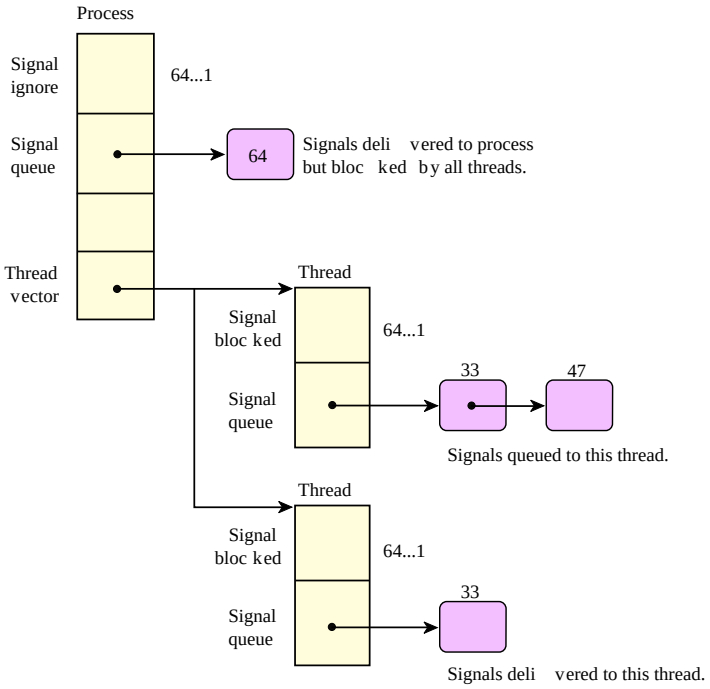
Microkernel call	POSIX call	Description
SignalKill()	*kill()*, *pthread_kill()*, *raise()*, *sigqueue()*	Set a signal on a process group, process, or thread.
SignalAction()	*sigaction()*	Define action to take on receipt of a signal.
SignalProcmask()	*sigprocmask()*, *pthread_sigmask()*	Change signal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

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

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

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

Figure 1Signal delivery.



The POSIX standard includes the concept of queued realtime signals. The QNX OS supports optional queuing of any signal, not just realtime signals. The queuing can be specified on a signal-by-signal basis within a process. Each signal can have an associated 8-bit code and a 64-bit value.

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

NOTE:
The kernel saves and restores the FPU context on entering and leaving signal handlers, so it's safe to use floating-point operations in them.

Page updated: August 11, 2025

Simple messages

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

Page updated: August 11, 2025

Special signals

QNX SDP | 8.0 | System Architecture | Developer | User

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

Their range is as follows:

Signal range	Description
1 ... 40	40 POSIX regular signals (including traditional UNIX signals)
41 ... 56	16 POSIX realtime signals (SIGRTMIN to SIGRTMAX)
57 ... 64	Eight special-purpose QNX OS signals, some of which are named (e.g., SIGSELECT). They're always masked, and attempts to unmask them are ignored.

The eight special signals cannot be ignored or caught. An attempt to call the *signal()* or *sigaction()* functions or the *SignalAction()* kernel call to change them will fail with an error of EINVAL. In addition, these signals are always blocked and have signal queuing enabled. An attempt to unblock these signals via the *sigprocmask()* function or *SignalProcmask()* kernel call will be quietly ignored.

Special signals are reserved for internal purposes. Currently, only one out of the possible eight is defined: SIGSELECT. It is used by *select()* to wait for I/O from multiple servers.

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

```
sigset_t *set;
struct sigaction action;

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

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

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

```
sigset_t *set;
siginfo_t info;

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

NOTE:
The above code is just an example; currently, the eighth special signal is undefined.

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

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

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

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

Page updated: August 11, 2025

Synchronous message passing

QNX SDP | 8.0 | System Architecture | Developer | User

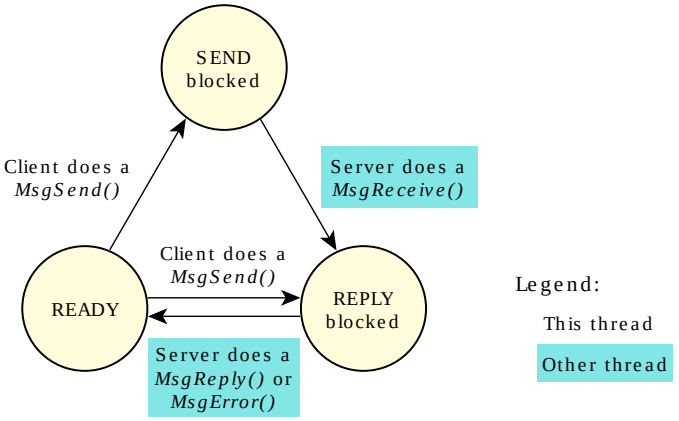
Synchronous messaging is the main form of IPC in the QNX OS.

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

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

Let's look first at the client thread:

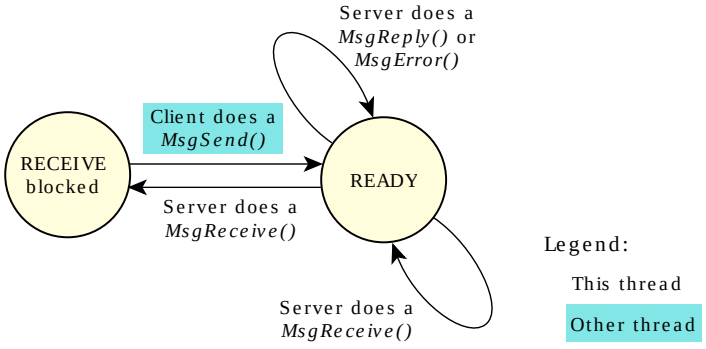
Figure 1Changes of state for a client thread in a send-receive-reply transaction.



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

Next, let's consider the server thread:

Figure 2Changes of state for a server thread in a send-receive-reply transaction.



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

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

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

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

MsgReply() vs *MsgError()*

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

Page updated: August 11, 2025

Typed memory

QNX SDP | 8.0 | System Architecture | Developer | User

POSIX typed memory provides an interface that lets you open named memory objects and perform mapping operations on them.

Typed memory is useful in providing an abstraction between BSP- or board-specific address layouts and device drivers or user code. You can reserve memory in advance for specific purposes, such as graphics, and remove that memory from the system RAM that programs typically allocate from.

POSIX specifies that typed memory pools (or objects) are created and defined in an implementation-specific fashion. Under QNX OS, typed memory objects are defined from the memory regions specified in the *asinfo* section of the system page. Thus, typed memory objects map directly to the address space hierarchy (*asinfo* segments) defined by startup. The typed memory objects also inherit the properties defined in *asinfo*, namely the physical address (or bounds) of the memory segments.

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

`memory`
: Physical addressability of the processor, which depends on the architecture.

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

`sysram`
: System RAM, which is memory that has been given to the OS to manage. This may also consist of multiple entries. The OS uses this pool for all general-purpose memory allocations on the system, including code, data, stack, heap, kernel and process manager data, shared memory objects, and memory-mapped files.

You can create additional entries, but only in startup, using the *as_add()* function (see the "Startup Library" chapter of *Building Embedded Systems*).

NOTE:
Typed memory objects can be either inside or outside SYSRAM. Objects that are inside SYSRAM are used to restrict normal allocations to a subset of the normal pool, while those outside SYSRAM are used to carve regions that aren't available for general-purpose allocation.

The names of typed memory regions are derived directly from the names of the *asinfo* segments. The *asinfo* section itself describes a hierarchy, and so the naming of typed memory objects is a hierarchy. Their names may contain intermediate slash (/) characters that are considered as path-component separators.

Let's look at a sample *asinfo* configuration, which you can display with the `pidin syspage=asinfo` command:

```
Header size=0x00000108, Total Size=0x00000cd0, #Cpu=2, Type=256
Section:asinfo offset:0x00000710 size:0x00000340 elsize:0x00000020
  0000) 0000000000000000-000000000000ffff o:ffff a:0000 p:100 c:0 n:/io
  0020) 0000000000000000-000fffffffffffff o:ffff a:0010 p:100 c:0
n:/memory
  0040) 0000000000000000-00000000ffffffff o:0020 a:0010 p:100 c:0
n:/memory/below4G
  0060) 0000000000000000-0000000000ffffff o:0020 a:0010 p:100 c:0
n:/memory/isa
  0080) 0000000006000000-00000000ffefffff o:0020 a:0013 p:100 c:0
n:/memory/device
  00a0) 00000000fff00000-00000000ffffffff o:0020 a:0005 p:100 c:0
n:/memory/rom
  00c0) 0000000000000000-000000000009f7ff o:0060 a:0017 p:100 c:0
n:/memory/isa/ram
  00e0) 0000000000100000-0000000000ffffff o:0060 a:0037 p:100 c:0
n:/memory/isa/ram
  0100) 0000000001000000-0000000005ffffff o:0040 a:0037 p:100 c:0
n:/memory/below4G/ram
  0120) 0000000006000000-000000003fedffff o:0080 a:0017 p:100 c:0
n:/memory/device/ram
  0140) 000000003ff00000-000000003fffffff o:0080 a:0017 p:100 c:0
n:/memory/device/ram
  0160) 00000000000f6a00-00000000000f6a23 o:0020 a:0007 p:100 c:0
n:/memory/acpi_rsdp
  0180) 00000000fee00000-00000000fee003ef o:0020 a:0003 p:100 c:0
n:/memory/lapic
```

For details about the above information, see the section on *asinfo* in the "System Page" chapter of *Building Embedded Systems*.

You use *posix_typed_mem_open()* to open a typed memory object, specifying the name of the object, the access mode, and flags that indicate how the object behaves when it's mapped. The name you pass to this function follows the above naming convention. If the name starts with a leading slash (`/`), an exact match is done. POSIX allows an implementation to define what happens when the name doesn't start with a leading slash; in QNX OS, a tail match is done on the pathname components specified.

You can also use ampersands and vertical bars (`&` and `|`) between segments to specify intersections and unions, respectively. You can specify an arbitrary number of them; they're evaluated from left to right, but parentheses aren't supported.

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

This name:	Resolves to:
`/memory`	`/memory`
`isa`	Fails because `isa` isn't a leaf in the hierarchy
`/memory/rom`	`/memory/rom`
`/sysram`	Fails because `sysram` isn't at the top of the hierarchy
`sysram`	All `sysram` segments, including four occurrences of `/memory/isa/ram/sysram`, two occurrences of `/memory/device/below4G/sysram`, and two occurrences of `/memory/device/ram/sysram`
`below4G&sysram`	The intersection of all physical address ranges for `below4G` with all physical address ranges for `sysram`. For example, if you have two memory regions, one being `/ram/below4G` with addresses from 0 to 4 GB, and the other being `/ram/sysram` with addresses from 1 GB to 6 GB, then `below4G&sysram` gives pages in the range 1 GB to 4 GB.

Specifying a union or intersection can be more useful than specifying a full name.

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

NOTE:
You can't open typed memory through the namespace interface (as you can for shared memory objects), because *posix_typed_mem_open()* requires an additional *tflag* argument, which you can't provide when you call *open()*.

If successful, *posix_typed_mem_open()* returns a file descriptor for the typed memory object. What you do with the file descriptor depends on which approach you want to take:

- the POSIX approach, using *mmap()* to establish a memory mapping from the typed memory object
- the QNX OS approach, using *shm_open()* and *shm_ctl()*. This isn't portable, but it's more efficient and secure.

Memory that's mapped from a typed-memory file descriptor is implicitly locked.

POSIX also defines *posix_typed_mem_get_info()*, which you can use to get information about a typed memory object. For more information about working with typed memory, see "Typed memory" in the "Working with Memory" chapter of the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

mmap()

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

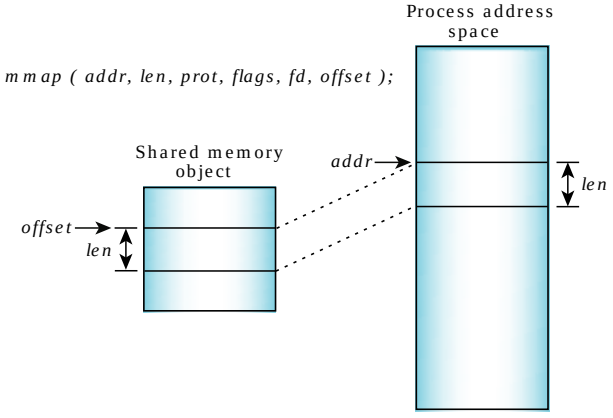
```
void * mmap( void *where_i_want_it,
             size_t length,
             int memory_protections,
             int mapping_flags,
             int fd,
             off_t offset_within_shared_memory );
```

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

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

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

Figure 1Mapping memory with *mmap()*.



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

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

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

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

NOTE:
On ARM targets, PROT_NOCACHE causes RAM to be mapped as normal noncached, but non-RAM to be mapped as strongly ordered device memory. For finer control, see *shm_ctl_special()*.

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

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

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

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

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

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

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

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

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

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

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

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

Or, it allocate a DMA buffer for a bus-mastering PCI network card:

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

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

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

Page updated: August 11, 2025

The QNX OS Microkernel

QNX SDP | 8.0 | System Architecture | Developer | User

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

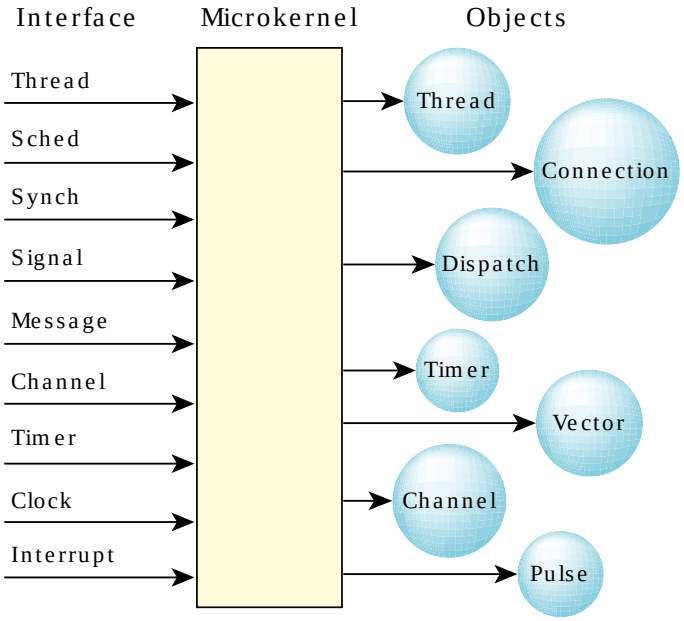
The POSIX features that aren't implemented in the `procnto` microkernel (file and device I/O, for example) are provided by optional processes and shared libraries.

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

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

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

Figure 1The microkernel.



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

Page updated: August 11, 2025

Barriers

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

```
#include <pthread.h>

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

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

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

```
#include <pthread.h>

int pthread_barrier_wait (pthread_barrier_t *barrier);
```

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

Here's an example:

```
/*
 *  barrier1.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <sys/neutrino.h>

pthread_barrier_t   barrier; // barrier synchronization object

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

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

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

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

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

In this release, the following barrier functions are included:

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

Page updated: August 11, 2025

Clock and timer services

QNX SDP | 8.0 | System Architecture | Developer | User

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

NOTE:
The kernel itself uses unsigned 64-bit numbers to count the nanoseconds since the start of January 1st, 1970. This design supports a time range that can be considered infinite for developers of QNX OS systems. Unlike with signed 32-bit time values, there's no risk of rollover in 2038 or, if unsigned values were used, in 2106.

The *ClockTime()* kernel call allows you to get or set the value of the system clock specified by an ID, including CLOCK_REALTIME and CLOCK_MONOTONIC, which maintain the system time. The system time is based on *ClockCycles()*, which returns the current value of a free-running 64-bit cycle counter, operating at the frequency read from the `SYSPAGE_ENTRY(qtime)->cycles_per_sec` field. For more information about this field and the other time-related fields in the system page data structure, go to the *qtime* reference in the "System Page" chapter in the *Building Embedded Systems* guide.

The *ClockCycles()* function is implemented on each processor architecture as a high-performance mechanism for timing short intervals. On x86 systems, *ClockCycles()* reads the Time Stamp Counter (RDTSC); on ARM systems, it reads the Generic Timer (CNTVCT_EL0 register).

NOTE:
QNX OS requires that the hardware underlying *ClockCycles()* be synchronized across all processors on a multicore system. If it isn't, you might encounter some unexpected behavior, such as drifting times and timers.

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

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

Page updated: August 11, 2025

CPU offlining

The CPU offlining feature allows a privileged application to stop `procnto` from using a CPU for scheduling threads.

> **NOTE:**
> Currently, the offlining and onlining scheduling feature is experimental.

This feature is useful for the following:

- Powering down a CPU (e.g., for lower workloads).
- Temporarily running software outside of the QNX OS (e.g., CPU diagnostics).
- Putting the system to sleep.

The privileged application runs a high priority thread (that can't be preempted) and tells `procnto` to stop using the CPU. This allows the thread to do its work outside of the QNX OS. Note that, when `procnto` isn't using the CPU, OS functionalities aren't available because any action that may cause the thread to context switch can make the system unstable.

To take a CPU offline, you must use `procnto`'s `-C` option to lower the clock IST to a maximum of 253. You need to leave a gap between the priorities of the clock IST and the IPI IST, which is 255.

The following steps describe the process of taking a CPU offline:

1. Assign the SCHED_OFFLINING scheduling policy to a thread via the *SchedSet()* call. The application must have the PROCMGR_AID_RUNSTATE ability enabled to set the policy.

   > **NOTE:**
   > An application needs a thread with the SCHED_OFFLINING policy set for every CPU being taken offline.

   The thread then increases its priority over that of the clock IST. The thread must also set its runmask to the one representing the CPU that it will take offline.

2. Offload the software timers queued on the CPU being taken offline to another CPU via a *TimerDelegate()* call with the _NTO_TIMER_DELEGATE action set. The application must delegate its timers to another CPU that isn't offline.

   > **NOTE:**
   > Some software timers may be ticking away on a queue owned by the CPU that you want to take offline. Therefore, they must be handed off to another CPU during the CPU offlining.

3. Instruct the scheduler to stop scheduling other threads on the CPU being taken offline via a *SchedCtl()* call with the SCHED_PROCESSOR_OFFLINE command set.

Once these steps are complete, the thread can run software outside of the QNX OS.

When the application is ready to give back the CPU to `procnto`, it needs to do the following:

1. Reclaim its timers via a *TimerDelegate()* call with the _NTO_TIMER_RECLAIM action set.
2. Mark the CPU as *online* via a *SchedCtl()* call with the SCHED_PROCESSOR_ONLINE command set.

Once these steps are complete, you can use the thread as usual and even use it to offline another CPU.

Page updated: August 11, 2025

Condvars: condition variables

QNX SDP | 8.0 | System Architecture | Developer | User

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

A condvar supports three operations:

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

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

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

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

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

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

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

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

NOTE:
An application shouldn't use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a *pthread_cond_wait()* or *pthread_cond_timedwait()* may not actually release the mutex (if it's been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

Page updated: August 11, 2025

External code invoked by the scheduler

QNX SDP | 8.0 | System Architecture | Developer | User

The scheduler may use code that's external to the kernel to perform certain functions. This code can affect thread scheduling and, thus, the deterministic behavior of the QNX OS.

The external code is sometimes in kernel callouts, which are stand-alone code segments that perform hardware-specific functions. For instance, the scheduler may invoke a callout to send an interprocessor interrupt (IPI) to force another processor in the system to make a scheduling decision. The callout implementation is hardware-specific, but it is expected that the hardware can send IPIs in constant time. During this time, another interrupt may occur, a high-priority thread may become unblocked (i.e., ready), or an asynchronous event such as a signal or timeout may result in a thread's cancellation. These occurrences can alter the subsequent scheduling decision.

For more information about kernel callouts, refer to "Kernel Callouts" in the *Building Embedded Systems* guide.

If tracing is enabled, the scheduler also invokes QNX OS tracing code (which is not in a kernel callout) when the associated thread state change occurs. Thread state events are emitted from the scheduler to preserve the correct event order. The tracing calls are expected to take constant time and have negligible impact on run time when tracing is off, but they can influence scheduling when tracing is on.

The *System Analysis Toolkit User's Guide* describes the QNX OS tracing mechanism and the various events related to scheduling and other system state changes.

Page updated: August 11, 2025

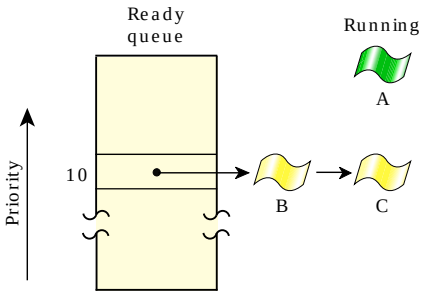
FIFO scheduling

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Figure 1FIFO scheduling.



Page updated: August 11, 2025

Design goals of the QNX OS

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

POSIX realtime and thread extensions

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

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

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

Page updated: August 11, 2025

Interrupt handling

QNX SDP | 8.0 | System Architecture | Developer | User

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. But, it is important to balance this with safety, predictable behaviour, and an avoidance of run-time allocation or queueing. QNX OS implements a low-latency path to schedule user threads to handle hardware interrupts.

When a hardware device asserts an interrupt, this will be processed by a Programmable Interrupt Controller (PIC), and if appropriate (not masked, not currently active), it will assert an interrupt on a CPU core. This will cause an exception, and the kernel's exception handler will execute. It will identify the interrupt, and for each thread associated with that interrupt, post an internal semaphore marking the thread as READY, mask the interrupt, and issue an end-of-interrupt (EOI) to the PIC. After the exception handler returns, unblocked thread(s) will be scheduled based on normal thread scheduling rules. A scheduled Interrupt Service Thread (IST) will generally deal with its hardware device, unmask the interrupt, and then block waiting for the next interrupt.

Page updated: August 11, 2025

IPC issues

QNX SDP | 8.0 | System Architecture | Developer | User

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

If only it were that simple!

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

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

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

Page updated: August 11, 2025

Interrupt calls

QNX SDP | 8.0 | System Architecture | Developer | User

The interrupt-handling API includes the kernel calls listed here.

Function	Description
InterruptAttachThread()	Attach a local function (an Interrupt Service Thread or IST) to an interrupt vector. This is the preferred call.
InterruptAttachEvent()	Generate an event on an interrupt, which will ready a thread.
InterruptDetach()	Detach from an interrupt using the ID returned by *InterruptAttachThread()* or *InterruptAttachEvent()*.
InterruptWait()	Wait for an interrupt.
InterruptEnable()	Enable hardware interrupts.
InterruptDisable()	Disable hardware interrupts.
InterruptMask()	Mask a hardware interrupt.
InterruptUnmask()	Unmask a hardware interrupt.

Using this API, a suitably privileged user-level thread can call *InterruptAttachThread()* or *InterruptAttachEvent()*, passing a hardware interrupt number to associate itself or an event with that interrupt number.

NOTE:

- The startup code is responsible for making sure that all interrupt sources are masked during system initialization. When the first call to *InterruptAttachThread()* or *InterruptAttachEvent()* is done for an interrupt vector, the kernel unmasks it. Similarly, when the last *InterruptDetach()* is done for an interrupt vector, the kernel remasks the level.
- You should disable interrupts for as little time as possible (i.e., the minimum time you need to access or deal with the hardware). Failure to do so may result in increased interrupt latency and the inability to meet realtime deadlines.
- Kernel calls and some library routines reenable interrupts. Masked interrupts aren't affected.

The following code sample shows how to attach a thread to an interrupt source:

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

#define IRQ_NUM 76

void* hardware_ist(void *unused)
{
    // Attach this thread to the interrupt source.
    int const id = InterruptAttachThread(IRQ_NUM,
_NTO_INTR_FLAGS_NO_UNMASK);
    if (id == -1) {
        // Handle errors.
    }

    for (;;) {
        // Wait for the interrupt to assert.
        // The FAST flag is more efficient, but cannot handle timeouts.
        int const rc = InterruptWait(_NTO_INTR_WAIT_FLAGS_UNMASK |
                                     _NTO_INTR_WAIT_FLAGS_FAST, NULL);
        if (rc == -1) {
            // Handle errors.
        }

        // Handle the interrupt. The interrupt is masked and will not
be asserted again
        // until InterruptWait() is called again with the UNMASK flag.
    }
}
```

With this approach, appropriately privileged user-level threads can dynamically attach to (and detach from) hardware interrupt vectors at runtime. These are regular OS threads, and thus can be debugged using regular source-level debug tools, and the priority of the work generated by hardware interrupts can be performed at OS-scheduled priorities rather than hardware-defined priorities.

For more information about interrupts, see the Interrupts chapter of *Getting Started with the QNX OS*, and the Handling Hardware Interrupts chapter of the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

Interrupt controller kernel module

QNX SDP | 8.0 | System Architecture | Developer | User

> **NOTE:**
> The interrupt controller kernel modules mentioned below are experimental

The interrupt controller kernel module transfers logic from startup-provided callouts to the kernel, allowing the kernel to make better interrupt handling decisions. This module allows for less contention on concurrent interrupt handling on multiple cores. This is in part due to the ability of the kernel module to make smarter decisions (e.g., when serializing access to the interrupt controller's resources) than the stock kernel that invokes opaque callouts provided by startup. A prime example is the clock interrupt, which can easily be triggered at the same time on multiple cores.

The interrupt controller modules depend on startup to set up the controller hardware and provide the matching `intrinfo` entries for the root controller and cascades. Currently, the modules only exist for aarch64 processors with a GICv2 or GICv3. GICv4 is backwards compatible with GICv3 and treated as such.

GICv2

The GICv2 module supports the following interrupts as separate entries:

- private peripheral interrupts (PPI) (which include software generated interrupts (SGI))
- shared peripheral interrupts (SPI)

The module depends on a startup that splits the PPIs and SPIs into different `intrinfo` entries. You can use the latest **libstartup** to build the startup.

To use the GICv2 module, add the following attribute to the `procnto` line in your buildfile:

```
[ module=gicv2 ]
```

GICv3

The GICv3 module supports the following interrupts as separate entries:

- private peripheral interrupts (PPI) (which include software generated interrupts (SGI))
- shared peripheral interrupts (SPI)
- external SPIs
- external PPIs
- locality-specific peripheral interrupts (LPI) via a translation table (ITS). In other words, the GICv3 module doesn't support direct LPIs.

> **NOTE:**
> All interrupts except PPIs are optional.

The CPU interface (GICC) is only supported via system registers (`ICC_xxx`), not via memory-mapped registers.

For LPIs, startup must provide the callouts for mask and unmask operations. LPI mask and unmask operation must use the ITS command queue for invalidating the configuration cache, which is a resource shared between cores.

For interprocessor interrupts (IPIs), startup must provide a value in the `SYSPAGE_CPU_ENTRY(aarch64, gic_map)->gic_cpu[]` array for each target CPU that the source CPU can write to `icc_sgi1r_el1`. This triggers the IPI on the target CPU.

To use the GICv3 module, add the following attribute to the `procnto` line in your buildfile:

```
[ module=gicv3 ]
```

Is your GIC module loaded?

To check if you have a GIC module loaded, enter:

```
# cat /proc/config
[...]
intrctlr:gicv2
```

Your system will display `intrctlr:gicv3` if a GICv3 module is loaded, or `intrctlr:N/A` if no module is loaded.

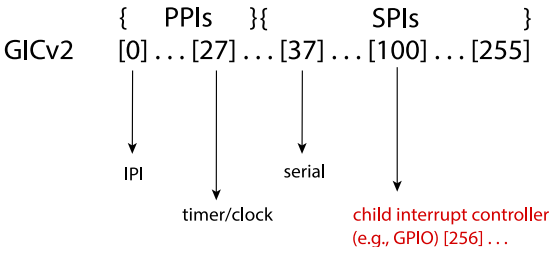
For more information about your GIC module, enter:

```
# cat /proc/ker/intr
gicv2                       /* module name */
2025/01/23-14:27:09EST      /* date of build */
spurious: 0                 /* number of spurious interrupts */
```

Cascades

It's possible to have a hierarchy of interrupt controllers, as shown in the image below:

Figure 1Hierarchy of interrupt controllers: child controller hooked to SPI 100



The modules see child controllers as opaque, and the controllers rely on startup to provide the callouts and configuration. Therefore, the modules treat these controllers as a shared resource when delivering their interrupts, and masking and unmasking them. For both GICv2 and GICv3 modules, child controllers should be hooked to an SPI.

The module supports all GENFLAG flags for child controllers, except for INTR_GENFLAG_ID_LOOP.

Caveats

The GIC modules only replaces the root controller, which matches several `intrinfo` entries (e.g., PPIs and SPIs). Each type of interrupt must be its own `intrinfo` entry.

Legacy GICv2 support in startup from **libstartup** bundles PPIs and SPIs in a single `intrinfo`. Ensure that your startup uses the latest **libstartup** to split them. For example:

```
Section:intrinfo offset:0x00000e68 size:0x000000c0 elsize:0x00000060
  0) vector_base:00000000, #vectors:32, cascade_vector:7fffffff
<--- PPIs: base: 0, 32 vectors
     cpu_intr_base:00000000, cpu_intr_stride:0, flags:0000,
local_stride:32768
      id => flags:0000, size:0030, rtn:ffffff80402050a8
     eoi => flags:0000, size:0018, rtn:ffffff80402050d8
     mask:ffffff80402050f0, unmask:ffffff8040205120,
config:ffffff8040205150
  1) vector_base:00000020, #vectors:128, cascade_vector:7fffffff
<--- SPIs: base: 32, n vectors
     cpu_intr_base:00000000, cpu_intr_stride:0, flags:0000,
local_stride:0
      id => flags:0000, size:0010, rtn:ffffff8040205160
     eoi => flags:0000, size:0018, rtn:ffffff8040205170
     mask:ffffff8040205188, unmask:ffffff80402051bc,
config:0000000000000000
```

GICv3 startups typically split the different interrupt types.

The INTR_GENFLAG_ID_LOOP flag isn't supported for cascades and the root controller won't use the flag for itself either.

Page updated: August 11, 2025

Interrupt latency

QNX SDP | 8.0 | System Architecture | Developer | User

Interrupt latency is the time between the assertion of the hardware interrupt and the execution of the first instruction after the IST returns from the kernel after blocking to wait for an interrupt.

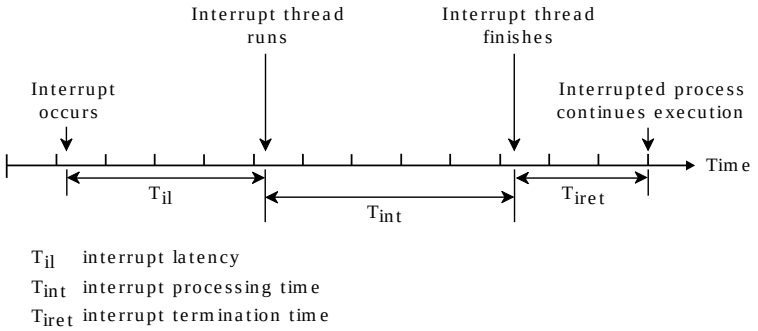
Interrupt latency comes from three components:

- Device to PIC - this is hardware bound and out of OS control
- PIC to exception handler - this is affected by the disabling of interrupts on the core to which the interrupt is routed, and the masking of the interrupt source.
- Exception handler to IST - this is affected by scheduling overhead, and scheduling configuration such as priority and prcoessor affinity.

The OS leaves interrupts fully enabled almost all the time, so the effect on interrupt latency is typically insignificant. But certain critical sections of code do require that interrupts be temporarily disabled.

Scheduling latency is the time between the last instruction of the kernel's exception handler and the execution of the first instruction of a driver thread. This usually means the time it takes to save the context of the currently executing thread and restore the context of the required driver thread. This time is also kept small in a QNX OS system.

Figure 1Interrupt thread simply terminates.



T_{il} interrupt latency
T_{int} interrupt processing time
T_{iret} interrupt termination time

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

Page updated: August 11, 2025

Thread lifecycle

QNX SDP | 8.0 | System Architecture | Developer | User

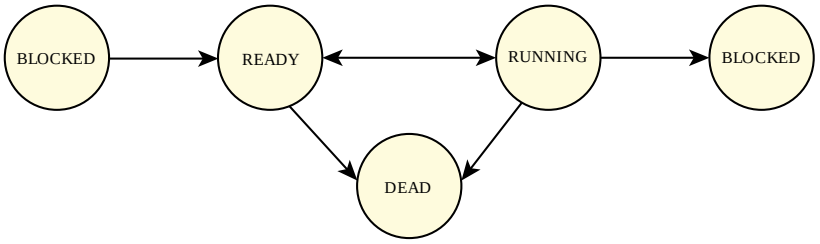
The number of threads within a process can vary widely, with threads being created and destroyed dynamically.

Thread creation (*pthread_create()*) involves allocating and initializing the necessary resources within the process's address space (e.g., thread stack) and starting the execution of the thread at some function in the address space. There are limits on how many threads you can create for a process; for details, see "Other system limits" in the QNX OS *User's Guide*.

Thread termination (*pthread_exit()*, *pthread_cancel()*) involves stopping the thread and reclaiming the thread's resources.

As a thread executes, its state can generally be described as either "blocked", "ready", or "running":

Figure 1Possible thread states and state transitions.



The term "blocked" covers the many different states in which a thread can't make progress (and hence, currently can't be executed). Most often, this is because it has issued a kernel call that has blocked, but it also occurs if another thread has signaled that the thread must be stopped. In this release, threads can block only themselves and not other threads, so a thread must run before going into a blocked state, including the STOPPED state. The exact state that a blocked thread is in depends on the reason that the thread blocked, such as the thread must wait for a message reply or for a semaphore to be posted.

When a thread terminates itself or gets terminated, encounters a fatal error, or receives a signal to shut down that it can't block or ignore (e.g., SIGKILL), it's then moved by the kernel to the DEAD state. The thread will never run again and the OS will reclaim its resources when convenient.

The exact possible thread states are:

STATE_BARRIER
The thread is blocked on a barrier (e.g., it called *pthread_barrier_wait()*).

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

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

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

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

STATE_MQ_RECEIVE
The thread is waiting on an empty queue after calling *mq_receive()*.

STATE_MQ_SEND
The thread is waiting on a full queue after calling *mq_send()*.

STATE_MUON_MUTEX
The thread is blocked on an internal kernel mutex.

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

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

STATE_READY
The thread is waiting to be executed while other threads are running.

STATE_RECEIVE

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

STATE_REPLY

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

STATE_RUNNING

The thread is being executed by a processor.

STATE_RWLOCK_READ

The thread is blocked trying to acquire a non-exclusive lock of a reader/writer lock after calling *pthread_rwlock_rdlock()*.

STATE_RWLOCK_WRITE

The thread is blocked trying to acquire an exclusive lock of a reader/writer lock after calling *pthread_rwlock_wrlock()*.

STATE_SEM

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

STATE_SEND

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

STATE_SIGSUSPEND

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

STATE_SIGWAITINFO

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

STATE_STOPPED

The thread has stopped, for example because it received a SIGSTOP or *ThreadCtl()* command.

STATE_WAITPAGE

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

NOTE:

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

Page updated: August 11, 2025

Manipulating priority and scheduling policies

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Page updated: August 11, 2025

Mutexes: mutual exclusion locks

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

Priority inheritance and mutexes

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

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

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

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

Other attributes

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

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

  After you set a mutex's priority ceiling, you should not try to lock that mutex from any process that has a priority that's greater than the mutex's priority ceiling. If you attempt this, the mutex locking will fail.

Note that robust mutexes and mutexes with priority ceilings use more system resources than other mutexes.

Page updated: August 11, 2025

Reader/writer locks

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

Page updated: August 11, 2025

Round-robin scheduling

QNX SDP | 8.0 | System Architecture | Developer | User

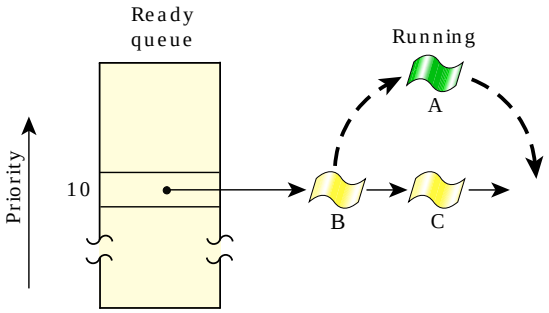
In round-robin scheduling, a thread selected to run continues executing until it:

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

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

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

Figure 1Round-robin scheduling.



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

Page updated: August 11, 2025

Thread scheduling

QNX SDP | 8.0 | System Architecture | Developer | User

When and how are scheduling decisions made?

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

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

- blocks
- is preempted
- yields

When does a thread block?

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

When is a thread preempted?

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

When does a thread yield?

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

Page updated: August 11, 2025

System services

QNX SDP | 8.0 | System Architecture | Developer | User

The microkernel has kernel calls to support the following:

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

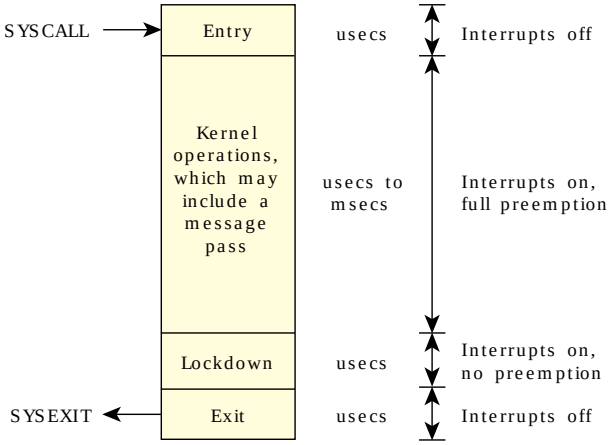
The entire OS is built upon these calls. The OS is fully preemptible, even while passing messages between processes; it resumes the message pass where it left off before preemption.

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

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

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

Figure 1QNX OS preemption details.



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

Page updated: August 11, 2025

Synchronization services

QNX SDP | 8.0 | System Architecture | Developer | User

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

The synchronization services include at least the following:

Synchronization service	Supported between processes
Mutexes	Yes[a]
Condvars	Yes
Barriers	Yes[a]
Reader/writer locks	Yes[a]
Semaphores	Yes
Send/Receive/Reply	Yes
Atomic operations	Yes

The above synchronization primitives are implemented directly by the kernel, except for atomic operations (which are implemented directly by the processor).

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

Page updated: August 11, 2025

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

Scheduling policies

QNX SDP | 8.0 | System Architecture | Developer | User

To meet the needs of various applications, the QNX OS provides these scheduling policies (or algorithms):

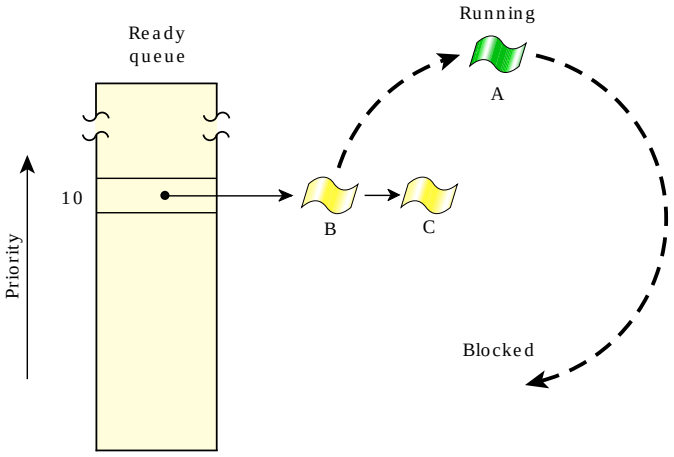
- FIFO scheduling
- round-robin scheduling
- sporadic scheduling

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

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

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

Figure 1Thread A blocks; Thread B runs.



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

For automating and optimizing scheduling configurations, you can integrate with MotionWise Schedule, an extension package for QNX SDP 8.0. The package provides deterministic scheduling capabilities to ensure that processes and tasks can meet their deadlines. For more information, refer to *MotionWise Schedule for QNX SDP 8.0*.

Page updated: August 11, 2025

Scheduling priority

QNX SDP | 8.0 | System Architecture | Developer | User

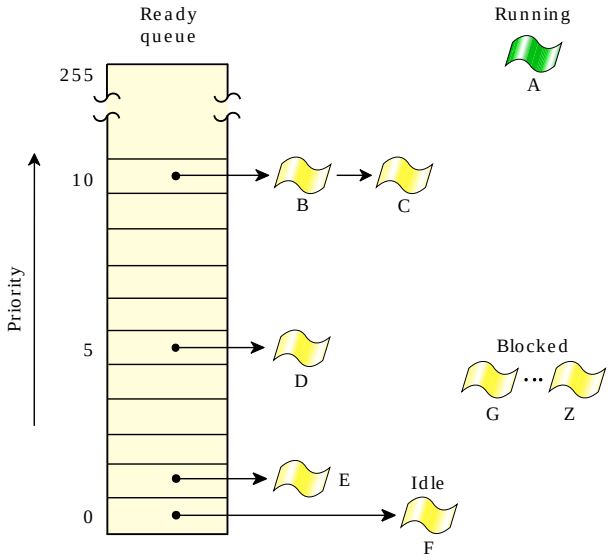
Every thread is assigned a priority. The thread scheduler selects the next thread to run by comparing the priorities of READY threads (i.e., those capable of using a processor).

Single-core ready queue

On a single-core system, the READY thread with the highest priority is selected to run.

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

Figure 1The ready queue in a single-core system.



Multicore (SMP) ready queue

On a multicore (SMP) system, the scheduler runs the highest-priority READY thread on one of the available processors (or CPUs, or cores). Additional processors run other threads in the system, though not necessarily the next highest-priority threads. Further details are given in the "Thread displacement on multicore systems" section of the *Programmer's Guide*.

For each scheduling decision on a given processor, the scheduler looks at the highest priority READY threads that can run on each cluster (i.e., group of associated processors) that the processor belongs to, and then selects the highest-priority thread from this set.

Threads on a cluster at the same priority are usually queued in FIFO, with the following exceptions:

- If a high-priority thread preempts the running thread, then it becomes the head of the thread list for its priority.
- When a thread changes priority. For more information, go to the *pthread_setschedprio()* entry in the QNX OS *C Library Reference*.

For information about clusters and configuring processor affinity for threads (i.e., associating threads with specific clusters), refer to the "Processor affinity, clusters, runmasks, and inherit masks" section of the QNX OS *Programmer's Guide*.

Scheduling priority levels

Every thread is assigned a priority. It can be explicitly set, but if it's not set explicitly, a thread inherits the priority of its parent thread by default.

The OS supports a total of 256 scheduling priority levels. Here's a summary, where *priority* is either 63 (the default) or a value you set with the `-P` option for `procnto` (see "Setting the upper end of the range of unprivileged priorities"):

Priority level	Owner

Priority level	Owner
0	Idle threads
1 through *priority* - 1	Unprivileged or privileged processes
priority through 253	Privileged processes with the PROCMGR_AID_PRIORITY ability (see *procmgr_ability()* in the *C Library Reference*)
254	Per-processor clock Interrupt Service Threads (ISTs) and other in-kernel ISTs
255	Per-processor InterProcessor (IPI) ISTs

NOTE:
To prevent *priority inversion*, the kernel may temporarily boost a thread's priority. For more information, see "Priority inheritance and mutexes" later in this chapter, and "Priority inheritance and messages" in the Interprocess Communication (IPC) chapter. The initial priority of the kernel's server threads is 10, but the first thing they all do is block in a *MsgReceive()*, so after that they operate at the priority of threads that send messages to them.

Setting the upper end of the range of unprivileged priorities

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

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

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

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

Setting the priority of a child process

When spawning a child process with a priority that's different than its parent, be aware that *posix_spawn()* and *spawn()* calls block until the initialization of the child process is complete. Because the child process's main thread performs the initialization at the priority set for the child process, it may create a priority inversion if that priority is lower than the parent.

Page updated: August 11, 2025

Semaphores

QNX SDP | 8.0 | System Architecture | Developer | User

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

Semaphores differ from other synchronization primitives in that they are "async safe" and can be manipulated by signal handlers. If the desired effect is to have a signal handler wake a thread, semaphores are the right choice.

The QNX OS implementation of semaphores supports both named and unnamed semaphores (see "Named and unnamed semaphores" below).

Semaphore or mutex?

A key advantage of semaphores over mutexes is that semaphores are defined to operate between processes.

Although our mutexes work between processes, the POSIX thread standard considers this an optional capability. Thus, if your code will rely on mutexes working between processes, to ensure portability you should probably use semaphores rather than mutexes.

However, depending on your needs you may wish to use mutexes rather than semaphores after considering the following:

- In general, mutexes are much faster than semaphores, which always require a kernel entry.
- For synchronization between threads in a single process, mutexes are more efficient than semaphores.
- Semaphores don't affect a thread's effective priority; if you need priority inheritance, use a mutex (see "Mutexes: mutual exclusion locks" in this chapter).

Using semaphores

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

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

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

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

For a complete list of functions you can use with semaphores, see the *sem_*()* functions in the *C Library Reference*.

Named and unnamed semaphores

Unnamed semaphores are created with *sem_init()*, and destroyed with *sem_destroy()*. They can be used by multiple threads within the same process. However, to use an unnamed semaphore between processes requires a shared memory object, which implies a large memory overhead.

Named semaphores are created with *sem_open()*, and closed with *sem_close()*. Like unnamed semaphores, they can be be used by multiple threads within the same process. Unlike unnamed semaphores, however, named semaphores can be easily shared between threads in different processes.

With the exception of anonymous named semaphores, named semaphores have a path and a name, so they can be found between processes without the use of shared memory objects and the overhead they imply. Named semaphores are therefore more useful for inter-process communication than are unnamed semaphores.

Additionally, named semaphores (including anonymous named semaphores) can be posted by a `sigevent` (e.g., attached to an IRQ or delivered by a server), and can be shared between a parent and a child process across a *fork()* without requiring a shared mapping.

Anonymous named semaphores don't have a path and name like other named semaphores, but they share the other advantages noted above. For more information about anonymous named semaphores, see *sem_open()* in the *C Library Reference*.

Page updated: August 11, 2025

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

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

The kernel rejects attempts to lock shared mutexes that would cause priority inheritance unless all mutex locking operations enter the kernel. This has a significant impact on the performance of shared mutexes that use the priority-inheritance protocol, but guarantees that noncooperating threads can't interfere with each other. All mutex operations on shared mutexes that support priority inheritance enter the kernel. In order to use a shared mutex in a safe manner without the performance overhead of entering the kernel systematically, you must disable priority inheritance for the mutex by using *pthread_mutexattr_setprotocol()* to set the PTHREAD_PRIO_NONE flag.

Page updated: August 11, 2025

Sporadic scheduling

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

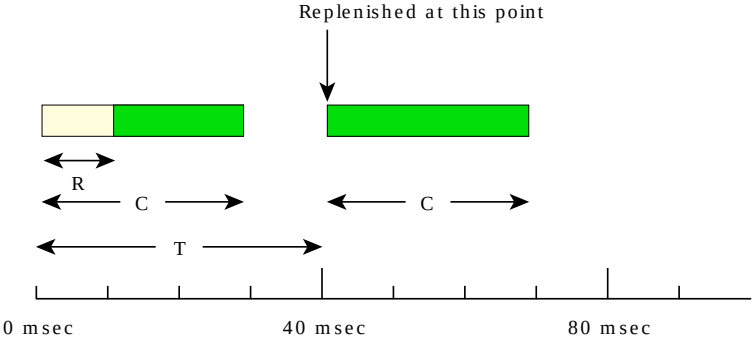
Replenishment period (T)
: The period of time during which a thread is allowed to consume its execution budget. To schedule replenishment operations, the POSIX implementation also uses this value as the offset from the time the thread becomes READY.

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

> **NOTE:**
> In a poorly configured system, a thread's execution budget may become eroded because of too much blocking—that is, it won't receive enough replenishments.

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

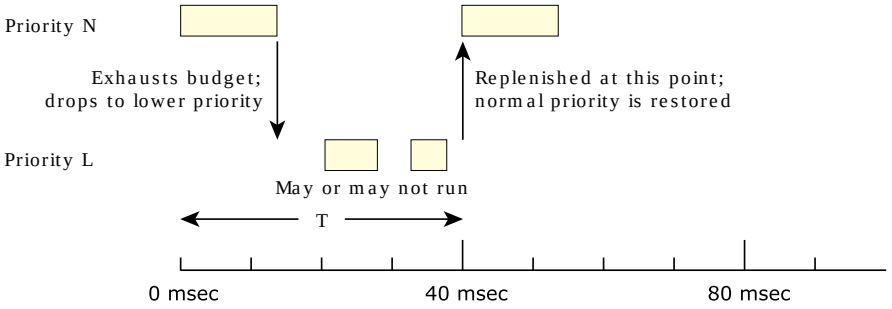
Figure 1A thread's budget is replenished periodically.



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

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

Figure 2A thread drops in priority until its budget is replenished.



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

Once the replenishment occurs, the thread's priority is raised to its original level. This guarantees that within a properly configured system, the thread will be given the opportunity *every period T* to run for a maximum execution time C. This ensures that a thread running at priority N will consume at most C/T of the system's CPU resources if there are other ready threads with a priority higher than L.

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

Page updated: August 11, 2025

Synchronization via atomic operations

QNX SDP | 8.0 | System Architecture | Developer | User

In some cases, you may want to perform a short operation (such as incrementing a variable) with the guarantee that the operation will perform *atomically*—that is, the operation won't be preempted by another thread or parallel execution on different cores in SMP.

The QNX OS provides atomic operations for:

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

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

Atomic operations are useful between two threads (SMP or single-processor).

Page updated: August 11, 2025

Synchronization via message passing

QNX SDP | 8.0 | System Architecture | Developer | User

Our Send/Receive/Reply message-passing IPC services implement an implicit synchronization by their blocking nature. These IPC services can, in many instances, render other synchronization services unnecessary.

For more information, see "Synchronous message passing" in the next chapter.

Page updated: August 11, 2025

Synchronization services implementation

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Page updated: August 11, 2025

Threads and processes

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

The following table lists the POSIX thread calls that have a corresponding microkernel thread call. The *pthread_** calls may have more functionalities implemented in the C library that are not provided by the kernel calls. Unless there are special circumstances, use the following *pthread_** calls instead of kernel calls:

POSIX call	Microkernel call	Description
pthread_create()	*ThreadCreate()*	Create a new thread of execution
pthread_exit()	*ThreadDestroy()*	Destroy a thread
pthread_detach()	*ThreadDetach()*	Detach a thread so it doesn't need to be joined
pthread_join()	*ThreadJoin()*	Join a thread waiting for its exit status

POSIX call	Microkernel call	Description
pthread_cancel()	*ThreadCancel()*	Cancel a thread at the next cancellation point
N/A	*ThreadCtl()*	Change a thread's QNX OS-specific thread characteristics
pthread_mutex_init()	*SyncTypeCreate()*	Create a mutex
pthread_mutex_destroy()	*SyncTypeDestroy()*	Destroy a mutex
pthread_mutex_lock()	*SyncMutexLock()*	Lock a mutex
pthread_mutex_trylock()	*SyncMutexLock()*	Conditionally lock a mutex
pthread_mutex_unlock()	*SyncMutexUnlock()*	Unlock a mutex
pthread_cond_init()	*SyncTypeCreate()*	Create a condition variable
pthread_cond_destroy()	*SyncTypeDestroy()*	Destroy a condition variable
pthread_cond_wait()	*SyncCondvarWait()*	Wait on a condition variable
pthread_cond_signal()	*SyncCondvarSignal()*	Signal a condition variable
pthread_cond_broadcast()	*SyncCondvarSignal()*	Broadcast a condition variable
pthread_getschedparam()	*SchedGet()*	Get the scheduling parameters and policy of a thread
pthread_setschedparam(), *pthread_setschedprio()*	*SchedSet()*	Set the scheduling parameters and policy of a thread
pthread_sigmask()	*SignalProcmask()*	Examine or set a thread's signal mask
pthread_kill()	*SignalKill()*	Send a signal to a specific thread

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

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

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

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

Page updated: August 11, 2025

Thread attributes

QNX SDP | 8.0 | System Architecture | Developer | User

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

Some of the more noteworthy thread-private resources are:

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

Priority
: Each thread has a priority that helps determine when it runs. A thread inherits its initial priority from its parent, but the priority can change, depending on the scheduling policy, explicit changes that the thread makes, or messages sent to the thread. For more information, see “Thread scheduling,” later in this chapter.

> **NOTE:**
> In the QNX OS, processes don't have priorities; their threads do.

Name
: You can assign a name to a thread; see the entries for *pthread_getname_np()* and *pthread_setname_np()* in the *C Library Reference*. Utilities such as `dumper` and `pidin` support thread names. Thread names are a QNX OS extension.

Register set
: Each thread has an instruction pointer (IP), stack pointer (SP), and other processor-specific register context.

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

Signal mask
: Each thread has its own signal mask.

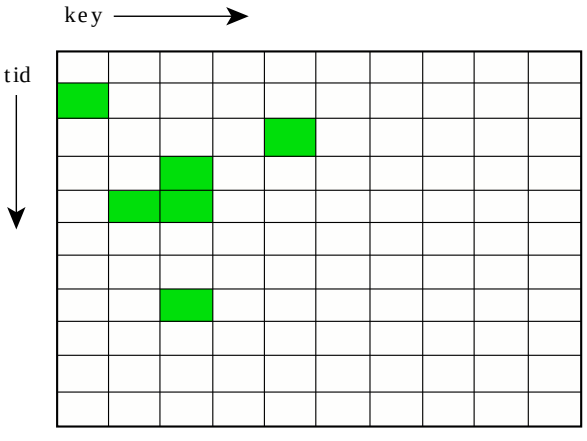
Thread local storage
: A thread has a system-defined data area called “thread local storage” (TLS). The TLS is used to store “per-thread” information (such as *tid*, *pid*, stack base, *errno*, and thread-specific key/data bindings). The TLS doesn't need to be accessed directly by a user application.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

Cancellation handlers
: A thread can have callback functions that are executed when it terminates.

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

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

Figure 1Sparse matrix (tid,key) to value mapping.



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

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

Page updated: August 11, 2025

Thread complexity issues

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

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

Page updated: August 11, 2025

Tracking time

QNX SDP | 8.0 | System Architecture | Developer | User

To keep QNX OS clocks on track, you have the following options:

ClockAdjust()

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

Page updated: August 11, 2025

Timers

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

Absolute date timers will be impacted by a change to the corresponding clock time whereas relative date and cyclical timers won't.

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

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

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

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

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

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

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

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

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

Microkernel call	POSIX call	Description
TimerAlarm()	*alarm()*	Set a process alarm

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

NOTE:
At most 50 timer events are generated per clock tick, to limit the consumption of system resources. If the system has a large number of timers set to expire in the same clock period, then at most 50 of the timer events will actually trigger on time; the rest will be handled on the next clock tick (subject to the same limitation of at most 50 timer events).

Page updated: August 11, 2025

Memory accounting

QNX SDP | 8.0 | System Architecture | Developer | User

Accounting for the memory used by your system and processes can be useful, especially when debugging. This section covers examples of how to read memory-related files and calculate reserved memory.

The actual calculations you perform will depend on your needs, and, due to the varying nature of different types of mapping, may be complex. The process counters discussed in this section provide a general value that represents reserved memory. To get an accurate value, you need to perform more complicated calculations using the raw information found in the memory manager.

Terminology

The following terminology is used for this discussion of memory accounting:

Anonymous
: Used to refer to memory that is not associated with an explicit object. Anonymous memory can come from anywhere in RAM and is zero-initialized when allocated.

Domain
: A collection of contiguous ranges of physical memory that serve a particular type of allocation (single page, multi-page, kernel).

Map
: A structure that is used to construct the page table entry for each page in the address space.

Page
: A fixed-length contiguous block of memory. The QNX OS process manager allocates memory in pages (typically 4 KB each).

Region
: A region is a sub-range of a process's address space. It is the result of a *mmap()* call. The pages in a region all share the same backing object, protection bits, and flags.

Page updated: August 11, 2025

Networking Architecture

QNX SDP | 8.0 | System Architecture | Developer | User

As with other service-providing processes in the QNX OS, the networking services execute outside the kernel. Developers are presented with a single unified interface, regardless of the configuration and number of networks involved.

This architecture allows:

- network drivers to be started and stopped *dynamically*
- multiple protocols to run together in any combination

Our native network subsystem consists of the network manager executable (`io-sock`), plus one or more shared library modules. These modules can include protocols and drivers (e.g., `devs-em.so`).

For more information, including an overview of the `io-sock` architecture and a discussion of the networking threading model, see the *High-Performance Networking Stack User's Guide*.

Page updated: August 11, 2025

Process Manager

QNX SDP | 8.0 | System Architecture | Developer | User

The Process Manager can create and manage multiple POSIX processes, each of which may contain multiple POSIX threads.

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

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

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

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

Page updated: August 11, 2025

File descriptor namespace

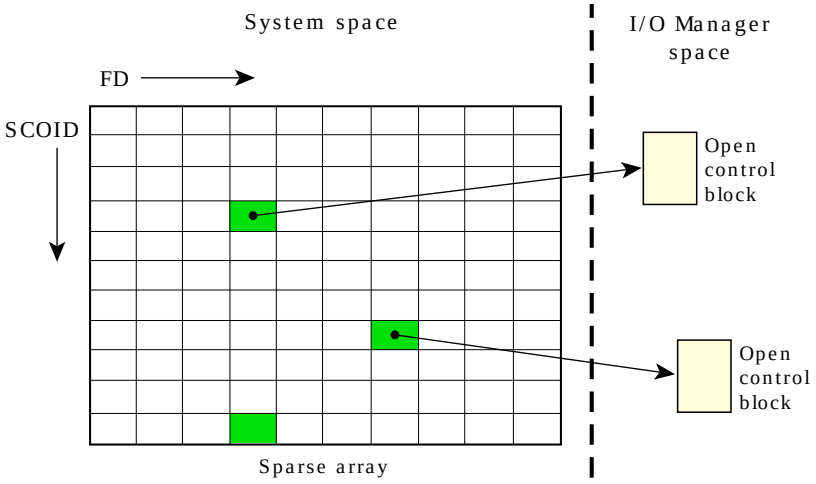
QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

Figure 1The SCOID and FD map to an OCB of an I/O Manager.



Page updated: August 11, 2025

Pathname management

QNX SDP | 8.0 | System Architecture | Developer | User

I/O resources *aren't* built into the microkernel, but are instead provided by resource manager processes that may be started dynamically at runtime. The `procnto` manager allows resource managers, through a standard API, to adopt a subset of the pathname space as a "domain of authority" to administer.

As other resource managers adopt their respective domains of authority, `procnto` becomes responsible for maintaining a pathname tree to track the processes that own portions of the pathname space. An adopted pathname is sometimes referred to as a "prefix" because it prefixes any pathnames that lie beneath it; prefixes can be arranged in a hierarchy called a *prefix tree*. The adopted pathname is also called a *mountpoint*, because that's where a server mounts into the pathname.

This approach to pathname space management is what allows QNX OS to preserve the POSIX semantics for device and file access, while making the presence of those services optional for small embedded systems.

At startup, `procnto` populates the pathname space with the following pathname prefixes:

Prefix	Description
/	Root of the filesystem.
/proc/boot/	Some of the files from the boot image presented as a flat filesystem.
/proc/*pid*	The running processes, each represented by its process ID (PID). For more information, see "Controlling processes via the **/proc** filesystem" in the Processes chapter of the QNX OS *Programmer's Guide*.
/dev/zero	A device that always returns zero. Used for allocating zero-filled pages using the *mmap()* function.
/dev/mem	A device that represents all physical memory.

Page updated: August 11, 2025

Process management

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Page updated: August 11, 2025

Process loading

QNX SDP | 8.0 | System Architecture | Developer | User

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

For an OS image, the code is executed in place. For other filesystem types, the code and data are loaded into main memory. In all cases, if the same process is loaded more than once, its code will be shared.

Page updated: August 11, 2025

Process primitives

QNX SDP | 8.0 | System Architecture | Developer | User

The process primitives include:

posix_spawn()
: POSIX

spawn()
: QNX OS

fork()
: POSIX

exec*()
: POSIX

Page updated: August 11, 2025

Process manager symbolic links

QNX SDP | 8.0 | System Architecture | Developer | User

We've discussed prefixes that map to a resource manager. Another form of mapping is a process manager symbolic link (procmgr symlink), which is a simple string substitution. Unlike a filesystem symbolic link, it exists only in memory instead of being a filesystem object.

Procmgr symlinks behave a lot like ordinary symlinks with the following exceptions:

- Because they don't exist in the filesystem, you can create them in cases where you can't create ordinary symlinks (e.g., read-only filesystems).
- To create them, a process requires the PATHSPACE ability. If security policies are in use, its type must have an `allow_link` rule that specifies the path where the symlink is created. Filesystem permissions are ignored.
- A procmgr symlink can permit a process whose root directory has been changed (i.e., a chrooted process) to access paths outside its root directory (for more information, go to "chroot (change root)" in the QNX OS *System Security Guide*).
- Procmgr symlinks are not persistent and are lost when the system reboots.

You can use the following methods to create procmgr symlinks:

- Using the `procmgr_symlink` command in a boot script (go to "Script files" in the `mkifs` entry in the *Utilities Reference*)
- Calling *pathmgr_symlink()*
- Using the `ln` (link) command. This command is typically used to create links on a filesystem: either hard links, or symbolic links by using the `-s` option. If you specify both `-s` and `-P`, then a procmgr symbolic link is created.

Command	Description
`ln -s` *existing_file symbolic_link*	Create a filesystem symbolic link.
`ln -Ps` *existing_file symbolic_link*	Create a procmgr symlink.

  Note that a procmgr symlink always takes precedence over a filesystem symbolic link.

As an example that uses `ln`, assume that you're running on a machine that doesn't have a local filesystem. However, there's a filesystem on another machine available over NFS (mounted at **/nfs_host**) that you wish to access as **/bin**. You accomplish this using the following procmgr symbolic link:

```
ln -Ps /nfs_host/x86_64/bin  /bin
```

This procmgr symlink causes **/bin** to be mapped into **/nfs_host/x86_64/bin**. For example, **/bin/ls** is replaced with the following: **/nfs_host/x86_64/bin/ls**

Page updated: August 11, 2025

Resolving pathnames

QNX SDP | 8.0 | System Architecture | Developer | User

When a process opens a file, the POSIX-compliant *open()* library routine first sends the pathname to `procnto`, where the pathname is compared against the prefix tree to determine which resource managers should be sent the *open()* message.

The prefix tree may contain identical or partially overlapping regions of authority—multiple servers can register the same prefix. If the regions are identical, the order of resolution can be specified (see "Ordering mountpoints," below). If the regions are overlapping, the responses from the path manager are ordered with the longest prefixes first; for prefixes of equal length, the same specified order of resolution applies as for identical regions.

For example, suppose we have these prefixes registered:

Prefix	Description
/	Power-Safe filesystem (`fs-qnx6.so`)
/dev/ser1	Serial device manager (`devc-ser*`)
/dev/ser2	Serial device manager (`devc-ser*`)
/dev/hd0	Raw disk volume (`devb-eide`)

The filesystem manager has registered a prefix for a mounted Power-Safe filesystem (i.e., **/**). The block device driver has registered a prefix for a block special file that represents an entire physical hard drive (i.e., **/dev/hd0**). The serial device manager has registered two prefixes for the two PC serial ports.

The following table illustrates the longest-match rule for pathname resolution:

This pathname:	matches:	and resolves to:
/dev/ser1	**/dev/ser1**	`devc-ser*`
/dev/ser2	**/dev/ser2**	`devc-ser*`
/dev/ser	**/**	`fs-qnx6.so`
/dev/hd0	**/dev/hd0**	`devb-eide.so`
/usr/jhsmith/test	**/**	`fs-qnx6.so`

Ordering mountpoints

Generally the order of resolving a filename is the order in which you mounted the filesystems at the same mountpoint (i.e., new mounts go on top of or in front of any existing ones). You can specify the order of resolution when you mount the filesystem. For example, you can use:

- the `before` and `after` keywords for block I/O (`devb-*`) drivers, in the `blk` options
- the `-Z b` and `-Z a` options to `fs-nfs3`

You can also use the `-o` option to `mount` with these keywords:

`before`
: Mount the filesystem so that it's resolved before any other filesystems mounted at the same pathname (in other words, it's placed in front of any existing mount). When you access a file, the system looks on this filesystem first.

`after`
: Mount the filesystem so that it's resolved after any other filesystems mounted at the same pathname (in other words, it's placed behind any existing mounts). When you access a file, the system looks on this filesystem last, and only if the file wasn't found on any other filesystems.

If you specify the appropriate `before` option, the filesystem floats in front of any other filesystems mounted at the same mountpoint, except those that you later mount with `before`. If you specify `after`, the filesystem goes behind any other filesystems mounted at the same mountpoint, except those that are already mounted with `after`. So, the search order for these filesystems is:

1. those mounted with `before`
2. those mounted with no flags
3. those mounted with `after`

with each list searched in order of mount requests. The first server to claim the name gets it. You would typically use `after` to have a filesystem wait at the back and pick up things the no one else is handling, and `before` to make sure a filesystem looks first at filenames.

Single-device mountpoints

Consider an example involving three servers:

Server A
: A Power-Safe filesystem. Its mountpoint is **/**. It contains the files **bin/true** and **bin/false**.

Server B
: A flash filesystem. Its mountpoint is **/bin**. It contains the files **ls** and **echo**.

Server C
: A single device that generates numbers. Its mountpoint is **/dev/random**.

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

Mountpoint	Server
/	Server A (Power-Safe filesystem)
/bin	Server B (flash filesystem)
/dev/random	Server C (device)

Of course, each “Server” name is actually an abbreviation for the *nd, pid, chid* for that particular server channel.

Now suppose a client wants to send a message to Server C. The client's code might look like this:

```
int fd;
fd = open("/dev/random", ...);
read(fd, ...);
close(fd);
```

In this case, the C library will ask the process manager for the servers that could *potentially* handle the path **/dev/random**. The process manager would return a list of servers:

- Server C (most likely; longest path match)
- Server A (least likely; shortest path match)

From this information, the library will then contact each server in turn and send it an *open* message, including the component of the path that the server should validate:

1. Server C receives a null path, since the request came in on the same path as the mountpoint.
2. Server A receives the path **dev/random**, since its mountpoint was **/**.

As soon as one server positively acknowledges the request, the library won't contact the remaining servers. This means Server A is contacted only if Server C denies the request.

This process is fairly straightforward with single device entries, where the first server is generally the server that will handle the request. Where it becomes interesting is in the case of *unioned filesystem mountpoints*.

Unioned filesystem mountpoints

Let's assume we have two servers set up as before:

Server A
: A Power-Safe filesystem. Its mountpoint is **/**. It contains the files **bin/true** and **bin/false**.

Server B
: A flash filesystem. Its mountpoint is **/bin**. It contains the files **ls** and **echo**.

Note that each server has a **/bin** directory, but with different contents.

Once both servers are mounted, you would see the following due to the union of the mountpoints:

/
: Server A

/bin
: Servers A and B

/bin/echo
: Server B

/bin/false
: Server A

/bin/ls
: Server B

/bin/true
: Server A

What's happening here is that the resolution for the path **/bin** takes place as before, but rather than limit the return to just one connection ID, *all* the servers are contacted and asked about their handling for the path:

```
DIR *dirp;
dirp = opendir("/bin", ...);
closedir(dirp);
```

This results in:

1. Server B receives a null path, since the request came in on the same path as the mountpoint.
2. Server A receives the path **"bin"**, since its mountpoint was **"/"**.

The result now is that we have a collection of file descriptors to servers who handle the path **/bin** (in this case two servers); the actual directory name entries are read in turn when a *readdir()* is called. If any of the names in the directory are accessed with a regular open, then the normal resolution procedure takes place and only one server is accessed.

For a more detailed look at this, see "Unioned filesystems" in the Resource Managers chapter of *Getting Started with the QNX OS*.

NOTE:
Running more than one resource manager on overlapping pathname spaces might cause deadlocks.

Why overlay mountpoints?

This overlaying of mountpoints is a very handy feature when doing field updates, servicing, etc. It also makes for a more unified system, where pathnames result in connections to servers regardless of what services they're providing, thus resulting in a more unified API.

Page updated: August 11, 2025

Memory management

QNX SDP | 8.0 | System Architecture | Developer | User

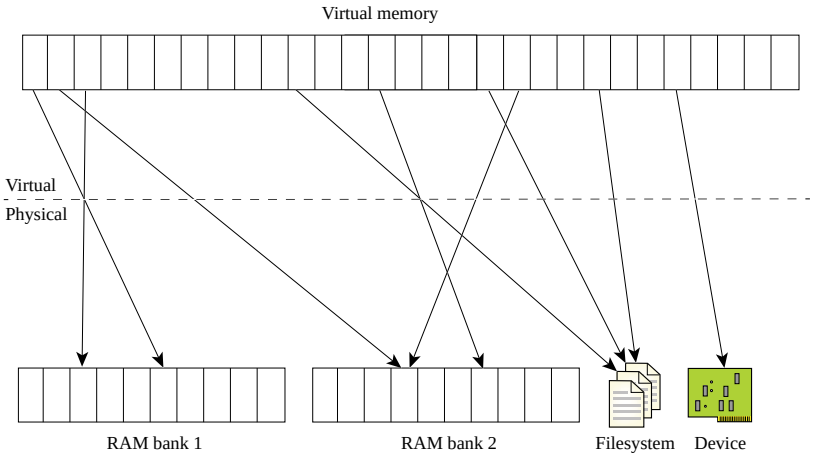
The virtual memory system implements the POSIX model for memory management along with some QNX OS extensions.

These extensions include:

- virtual address spaces that contain mappings to memory objects
- mappings that can be manipulated at page-size granularity
- memory-mapped files that use persistent filesystem backing storage
- shared memory objects that use RAM
- shared memory objects using the *shm_ctl()* and *shm_ctl_special()* interface
- typed memory objects that represent physical memory ranges
- MAP_ANON to allocate anonymous memory (memory that isn't associated with a file or device)

The memory manager's task is to create an abstraction of all components in the system that present themselves as physical memory devices. This abstraction allows the content of memory to come from anywhere in RAM, files in a filesystem, mapped device registers, and other types of memory such as read-only memory (ROM) or non-volatile memory (NVRAM). Virtual memory gives a per-process view of memory, which helps with safety and security, but also allows for shared memory objects and shared libraries.

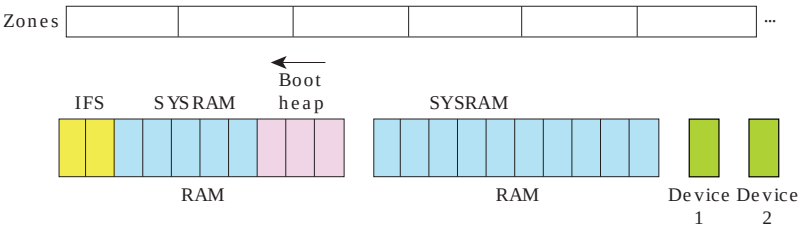
Figure 1Virtual memory is an abstraction of physical memory.



The hardware that you run QNX OS on must have a Memory Management Unit (MMU). The MMU is turned on early in the startup sequence, before the kernel itself starts. Once the MMU has started, there are no direct references to physical memory; the MMU translates virtual addresses into physical ones. It's up to the memory manager to create the information for managing physical memory, including the mappings of virtual to physical addresses.

The MMU divides physical memory into fixed-size pieces called *pages* that are usually (but not always) 4 KB. Some physical memory is used for the image filesystem (IFS), and the memory manager has a boot heap that it uses to allocate the data structures that it needs. System RAM (or “sysram”) is the physical memory that the memory manager can allocate in response to requests for memory.

Figure 2Managing physical memory.



A *zone* is a fixed-size, contiguous range of physical memory (currently 16 MB). The division of RAM into zones is intended to reduce fragmentation by having zones dedicated to specific allocation types.

Each process has its own virtual address space, which is a per-process view of virtual memory. This protects process address spaces from each other, preventing coding errors in a thread in one process from damaging memory used by threads in other processes or even in the OS. This protection is useful both for development and

for the installed runtime system, because it makes postmortem analysis possible. Almost all OS services and drivers are independent processes.

A process's virtual address space is divided into *regions*, which are ranges of contiguous virtual addresses that refer to various pieces of memory. *Maps* associate virtual addresses with physical addresses, and include permissions. The *mmap()* function creates regions and the maps.

All memory regions are fully backed when they're created. The memory manager keeps track of how much memory is needed to back all allocated memory. The *mmap()* function fails if there isn't enough memory for it to reserve.

Locking memory

The QNX OS supports POSIX memory locking APIs, but only for compatibility reasons. These calls will silently succeed, but in this release all memory is “superlocked”, meaning:

- faulting occurs only when the virtual address isn't mapped or the process doesn't have the necessary access permissions; it always results in a failure of the operation
- all memory must be initialized and privatized, and the permissions set, as soon as the memory is mapped

In this release, the MAP_LAZY flag is sliently ignored and has no effect on mapping behavior.

Page updated: August 11, 2025

Resource Managers

QNX SDP | 8.0 | System Architecture | Developer | User

To give the QNX OS a great degree of flexibility, to minimize the runtime memory requirements of the final system, and to cope with the wide variety of devices that may be found in a custom embedded system, the OS allows user-written processes to act as *resource managers* that can be started and stopped dynamically.

Resource managers are typically responsible for presenting an interface to various types of devices. This may involve managing actual hardware devices (like serial ports, parallel ports, network cards, and disk drives) or virtual devices (like **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers*. But unlike device drivers, resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program.

Page updated: August 11, 2025

Message types

QNX SDP | 8.0 | System Architecture | Developer | User

Architecturally, there are two categories of messages that a resource manager will receive:

- *connect messages*
- *I/O messages*

A connect message is issued by the client to perform an operation based on a pathname (e.g., an `io_open` message). This may involve performing operations such as permission checks (does the client have the correct permission to open this device?) and setting up a *context* for that request.

An I/O message is one that relies upon this context (created between the client and the resource manager) to perform subsequent processing of I/O messages (e.g., `io_read`).

There are good reasons for this design. It would be inefficient to pass the full pathname for each and every *read()* request, for example. The `io_open` handler can also perform tasks that we want done only once (e.g., permission checks) rather than with each I/O message. Also, when the *read()* has read 4096 bytes from a disk file, there may be another 20 megabytes still waiting to be read. Therefore, the *read()* function would need to have some context information telling it the position within the file it's reading from.

Page updated: August 11, 2025

Communication via native IPC

QNX SDP | 8.0 | System Architecture | Developer | User

Once a resource manager has established its pathname prefix, it will receive messages whenever any client program tries to do an *open()*, *read()*, *write()*, etc. on that pathname.

For example, after `devc-ser*` has taken over the pathname **/dev/ser1**, and a client program executes:

```
fd = open ("/dev/ser1", O_RDONLY);
```

the client's C library will construct an `io_open` message, which it then sends to the `devc-ser*` resource manager via IPC.

Some time later, when the client program executes:

```
read (fd, buf, BUFSIZ);
```

the client's C library constructs an `io_read` message, which is then sent to the resource manager.

A key point is that *all* communications between the client program and the resource manager are done through *native IPC messaging*. This allows for a number of unique features:

- A well-defined interface to application programs. In a development environment, this allows a very clean division of labor for the implementation of the client side and the resource manager side.
- A simple interface to the resource manager. Since all interactions with the resource manager go through native IPC, and there are no special “back door” hooks or arrangements with the OS, the writer of a resource manager can focus on the task at hand, rather than worry about all the special considerations needed in other operating systems.

NOTE:
All QNX OS device drivers and filesystems are implemented as resource managers. This means that everything that a “native” QNX OS device driver or filesystem can do, a user-written resource manager can do as well.

Consider FTP filesystems, for instance. Here a resource manager would take over a portion of the pathname space (e.g., **/ftp**) and allow users to `cd` into FTP sites to get files. For example, `cd /ftp/rtfm.mit.edu/pub` would connect to the FTP site **rtfm.mit.edu** and change directory to **/pub**. After that point, the user could open, edit, or copy files.

Application-specific filesystems would be another example of a user-written resource manager. Given an application that makes extensive use of disk-based files, a custom tailored filesystem can be written that works with that application and delivers superior performance.

The possibilities for custom resource managers are limited only by the application developer's imagination.

Page updated: August 11, 2025

Resource manager architecture

QNX SDP | 8.0 | System Architecture | Developer | User

Here is the heart of a resource manager:

```
initialize the dispatch interface
register the pathname with the process manager
DO forever
    receive a message
    SWITCH on the type of message
        CASE io_open:
            perform io_open processing
            ENDCASE
        CASE io_read:
            perform io_read processing
            ENDCASE
        CASE io_write:
            perform io_write processing
            ENDCASE
        .   // etc. handle all other messages that may occur,
        .   // performing processing as appropriate
    ENDSWITCH
ENDDO
```

The architecture contains three parts:

1. A channel that is created so that client programs can connect to the resource manager to send it messages.
2. The pathname (or pathnames) that the resource manager is going to be responsible for. This pathname is registered with the process manager so that it can resolve open requests for that particular pathname to this resource manager.
3. A loop that receives and processes messages.

This message-processing structure (the switch/case, above) is required for each and every resource manager. However, we provide a set of convenient library functions to handle this functionality (and other key functionality as well).

Page updated: August 11, 2025

The types of resource managers

QNX SDP | 8.0 | System Architecture | Developer | User

Depending on how much work you want to do yourself in order to present a proper POSIX filesystem to the client, you can break resource managers into two types:

- Device resource managers
- Filesystem resource managers

Page updated: August 11, 2025

The resource manager shared library

QNX SDP | 8.0 | System Architecture | Developer | User

In a custom embedded system, part of the design effort may be spent writing a resource manager, because there may not be an off-the-shelf driver available for the custom hardware component in the system.

Our resource manager shared library makes this task relatively simple.

Page updated: August 11, 2025

Summary

QNX SDP | 8.0 | System Architecture | Developer | User

By supporting pathname space mapping, by having a well-defined interface to resource managers, and by providing a set of libraries for common resource manager functions, the QNX OS offers the developer unprecedented flexibility and simplicity in developing “drivers” for new hardware—a critical feature for many embedded systems.

For more details on developing a resource manager, see the Resource Managers chapter of *Getting Started with the QNX OS*, and the *Writing a Resource Manager* guide.

Page updated: August 11, 2025

How do client programs talk to resource managers?

QNX SDP | 8.0 | System Architecture | Developer | User

Since the QNX OS is a distributed, microkernel OS with virtually all nonkernel functionality provided by user-installable programs, a clean and well-defined interface is required between client programs and resource managers.

All resource manager functions are documented; there's no “magic” or private interface between the kernel and a resource manager. In fact, a resource manager is basically a user-level server program that accepts messages from other programs and, optionally, communicates with hardware. Again, the power and flexibility of our native IPC services allow the resource manager to be decoupled from the OS.

The binding between the resource manager and the client programs that use the associated resource is done through a flexible mechanism called *pathname space mapping*.

In pathname space mapping, an association is made between a pathname and a resource manager. The resource manager sets up this pathname space mapping by informing the process manager that it is the one responsible for handling requests at (or below, in the case of filesystems) a certain *mountpoint*. This allows the process manager to associate services (i.e., functions provided by resource managers) with pathnames.

For example, a serial port may be managed by a resource manager called `devc-ser*`, but the actual resource may be called **/dev/ser1** in the pathname space. Therefore, when a client program requests serial port services, it typically does so by opening a serial port—in this case **/dev/ser1**.

Page updated: August 11, 2025

Why write a resource manager?

QNX SDP | 8.0 | System Architecture | Developer | User

Here are a few reasons why you may want to write a resource manager:

- The client API is POSIX.

  The API for communicating with the resource manager is for the most part, POSIX. All C programmers are familiar with the *open()*, *read()*, and *write()* functions. Training costs are minimized, and so is the need to document the interface to your server.

- You can reduce the number of interface types.

  If you have many server processes, writing each server as a resource manager keeps the number of different interfaces that clients need to use to a minimum.

  For example, suppose you have a team of programmers building your overall application, and each programmer is writing one or more servers for that application. These programmers may work directly for your company, or they may belong to partner companies who are developing addon hardware for your modular platform.

  If the servers are resource managers, then the interface to all of those servers is the POSIX functions: *open()*, *read()*, *write()*, and whatever else makes sense. For control-type messages that don't fit into a read/write model, there's *devctl()* (although *devctl()* isn't POSIX).

- Command-line utilities can communicate with resource managers.

  Since the API for communicating with a resource manager is the POSIX set of functions, and since standard POSIX utilities use this API, you can use the utilities for communicating with the resource managers.

  For instance, suppose a resource manager registers the name **/proc/my_stats**. If you open this name and read from it, the resource manager responds with a body of text that describes its statistics.

  The `cat` utility takes the name of a file and opens the file, reads from it, and displays whatever it reads to standard output (typically the screen). As a result, you could type:

  ```
  cat /proc/my_stats
  ```

  and the resource manager would respond with the appropriate statistics.

  You could also use command-line utilities for a robot-arm driver. The driver could register the name, **/dev/robot/arm/angle**, and interpret any writes to this device as the angle to set the robot arm to. To test the driver from the command line, you'd type:

  ```
  echo 87 >/dev/robot/arm/angle
  ```

  The `echo` utility opens **/dev/robot/arm/angle** and writes the string ("87") to it. The driver handles the write by setting the robot arm to 87 degrees. Note that this was accomplished without writing a special tester program.

  Another example would be names such as **/dev/robot/registers/r1**, **r2**, and so on. Reading from these names returns the contents of the corresponding registers; writing to these names sets the corresponding registers to the given values.

  Even if all of your other IPC is done via some non-POSIX API, it's still worth having one thread written as a resource manager for responding to reads and writes for doing things as shown above.

Page updated: August 11, 2025

TCP/IP Networking

QNX SDP | 8.0 | System Architecture | Developer | User

Our TCP/IP stack is included in the `io-sock` networking stack and uses the common FreeBSD API.

For more information on the Internet protocols ported from FreeBSD, go to the "Protocols" chapter in the *High-Performance Networking Stack User's Guide*.

Page updated: August 11, 2025

Thread Local Storage (TLS)

QNX SDP | 8.0 | System Architecture | Developer | User

In multithreaded processes, all of the threads in the process share the same static and global variables. Thread local storage (TLS) allows each thread in a process to store its own copy of static or global memory. When using TLS, memory for the unique data is allocated when the thread starts and deallocated when it ends. Any pointers to variables in threads using TLS become invalid when the thread ends. You can declare the following keywords with a variable to provide it with TLS duration:

- `_Thread_local` (in C11 and later)
- `thread_local` (in C++11 and later)
- `__thread` (a GCC extension)

NOTE:

- In C11 and later, the **<threads.h>** header file defines `thread_local` as a macro to use as an alias for the `_Thread_local` keyword.
- For better code portability, avoid using the `__thread` keyword, as it's a GCC extension and may not be supported on all systems.

You can set these keywords with the following variable types:

- global
- file-scoped static
- function-scoped static

However, you can't set them on block-scoped automatic, or non-static data members. These keywords can be used in library code and the executable. For example, use the `thread_local` keyword to declare a global variable:

```
thread_local unsigned global_x;
```

Use the `thread_local` keyword to declare a file-scoped static variable:

```
static thread_local unsigned file_x;
```

Use the `thread_local` keyword to declare a function-scoped static variable:

```
unsigned update_thread_count(unsigned value) {
    static thread_local unsigned count;
    count += value; /* Each thread will have its own value for count */
    return count;
}
```

TLS functionality isn't equivalent to using thread-specific calls, such as *pthread_setspecific()*, which use thread-specific data instead. These calls are less efficient in the way that they store and access data. Thread-specific calls take more work to set up because a combination of these calls is required for every thread that needs data. For more information on thread-specific data, refer to the "Thread attributes" page.

Page updated: August 11, 2025

The Microkernel Instrumentation

QNX SDP | 8.0 | System Architecture | Developer | User

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

The `procnto-smp-instr` instrumentation uses very little overhead and gives exceptionally good performance. The additional amount of code for the instrumentation is a relatively small price to pay for the added power and flexibility of this useful tool.

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

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

Page updated: August 11, 2025

Data interpretation

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

Page updated: August 11, 2025

Event control

QNX SDP | 8.0 | System Architecture | Developer | User

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

Specifically, you can:

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

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

Page updated: August 11, 2025

Modes of emission

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

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

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

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

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

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

Page updated: August 11, 2025

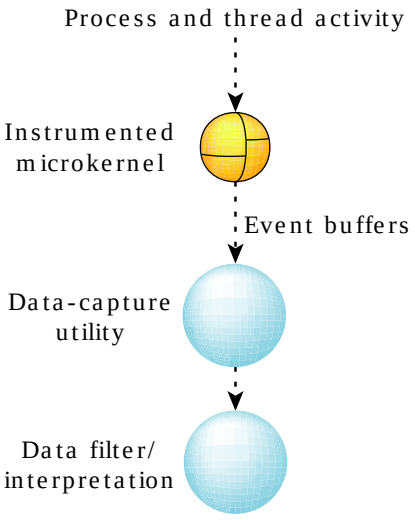
Instrumentation at a glance

QNX SDP | 8.0 | System Architecture | Developer | User

Here are the essential tasks involved in kernel instrumentation:

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

Figure 1Instrumentation at a glance.



Page updated: August 11, 2025

Proactive tracing

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

First, one process must register itself as the designated trace-logging process by issuing the `_NTO_TRACE_LOGGER_ATTACH` command. This command has *start_ev* and *stop_ev* parameters, which can be NULL or pointers to sigevents that are to be delivered when tracing starts and stops:

```
TraceEvent(_NTO_TRACE_LOGGER_ATTACH, start_ev, stop_ev);
```

NOTE:
Unlike in past releases, only one process can perform proactive tracing at a given time.

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

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

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

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

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

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

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

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

    return 0;
}
```

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

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

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

Page updated: August 11, 2025

Ring buffer

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Page updated: August 11, 2025

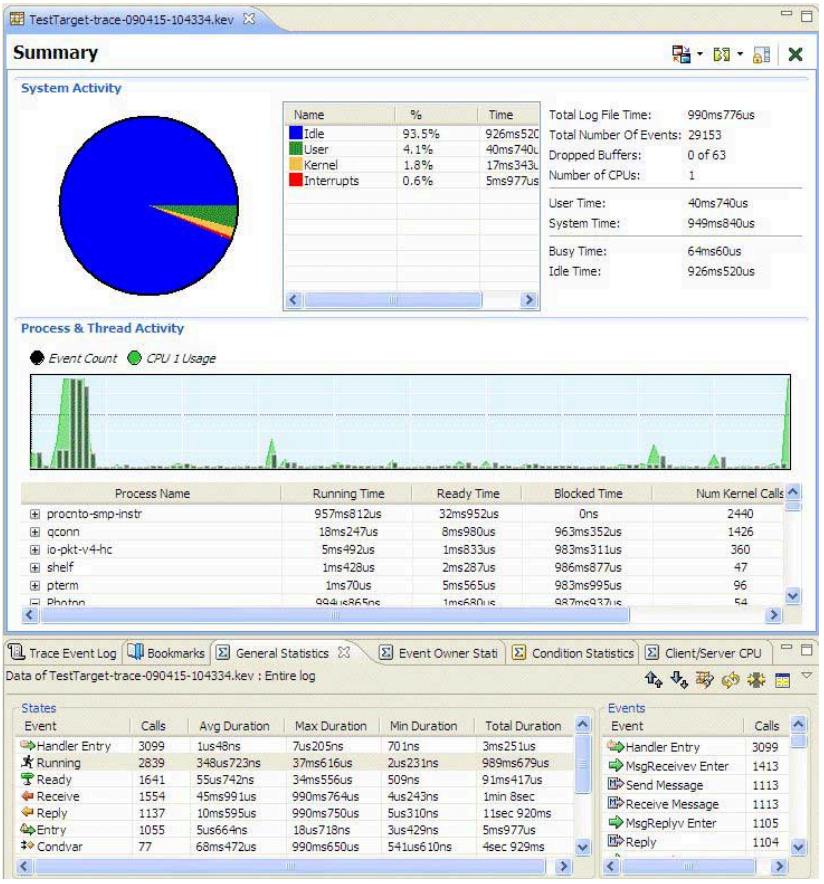
System analysis with the IDE

QNX SDP | 8.0 | System Architecture | Developer | User

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

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

Figure 1The IDE helps you visualize system activity.



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

Page updated: August 11, 2025

What is Real Time and Why Do I Need It?

QNX SDP | 8.0 | System Architecture | Developer | User

Real time is an often misunderstood—and misapplied—property of operating systems. This appendix provides a summary of some of the critical elements of realtime computing and discusses a few design considerations and benefits.

We can start with a basic definition of a realtime system, as defined in the FAQ for the `comp.realtime` newsgroup:

> " A realtime system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred. "

Real time, then, is a property of systems where time is literally "of the essence." In a realtime system, the value of a computation depends on how timely the answer is. For example, a computation that's completed late has a diminishing value, or no value whatsoever, and a computation completed early is of no extra value. Real time is always a matter of degree, since even batch computing systems have a realtime aspect to them—nobody wants to get their payroll deposit two weeks late!

Problems arise when there's competition for resources in the system, and resources are shared among many activities, which is where we begin to apply the realtime property to operating systems. A critical step in implementing any realtime system is the determination of a schedule of activities such that all activities are completed on time.

Any realtime system comprises different types of activities:

- those that can be scheduled
- those that can't be scheduled, such as operating-system facilities
- non-realtime activities

If non-schedulable activities can execute in preference to schedulable activities, they'll affect the ability of the system to handle time constraints.

Hard and soft real time

Hard real time is a property of the timeliness of a computation in the system. A hard realtime constraint in the system is one for which there's no value to a computation if it's late, and the effects of a late computation may be catastrophic to the system. Simply put, a hard realtime system is one where all of the activities must be completed on time.

Soft real time is a property of the timeliness of a computation where the value diminishes according to its tardiness. A soft realtime system can tolerate some late answers to soft realtime computations, as long as the value hasn't diminished to zero. A soft realtime system often carries meta requirements, such as a stochastic model of acceptable frequency of late computations. This is very different from conventional applications of the term, which don't account for how late a computation is completed or how frequently this may occur.

Soft real time is often improperly applied to operating systems that don't satisfy the necessary conditions for guaranteeing that computations can be completed on time. Such operating systems are best described as quasi-realtime or pseudo-realtime in that they execute realtime activities in preference to others whenever necessary, but don't adequately account for non-schedulable activities in the system.

Who needs real time?

Traditionally, realtime operating systems have been used in "mission-critical" environments requiring hard realtime capability, where failure to perform activities in a timely manner can result in harm to persons or property.

Often overlooked, however, are situations where there's a need to meet quality of service guarantees, particularly when failure to do so could result in financial penalty. This covers obvious service scenarios, such as "thirty minutes or it's free," but it also includes intangible penalties, such as lost opportunities or loss of market share.

More and more, real time is being employed in consumer devices—complex systems that demand the utmost in reliability. For example, a non-realtime device aimed at presenting live video, such as MPEG movies, that depends on software for any part of the delivery of the content, may experience dropped frames at a rate that the customer perceives as unacceptable.

In designing systems, developers need to assess whether the performance benefits warrant the use of realtime technology. A decision made early on can have unforeseen consequences when overload of the deployed system leads to pathological behavior in which most or none of the activities complete on time, if at all.

Realtime technology can be applied to conventional systems in ways that have a positive impact on the user experience, either by improving the perceived response to certain events, or ensuring that important activities execute preferentially with respect to others in the system.

What is a realtime OS?

Our definition of what constitutes a hard realtime operating system is based on realtime scheduling theory that's consistent with industry practice:

> " A hard realtime operating system must guarantee that a feasible schedule can be executed, given sufficient computational capacity if external factors are discounted. External factors, in this case, are devices that may generate interrupts, including network interfaces that generate interrupts in response to network traffic. "

In other words, if a system designer controls the environment of the system, the operating system itself won't be the cause of any tardy computations. We can apply this term to conventional operating systems—which typically execute tasks according to their priority—by referring to scheduling theory and deriving a minimum set of conditions that must be met. In brief, scheduling theory demonstrates that a schedule can be translated into static priority assignments in a way that guarantees timeliness. It does so by dividing the time available into periodic divisions and assuming a certain proportion of each division is reserved for particular realtime activities.

In order to do so, the following basic requirements must be met:

1. Higher-priority tasks always execute in preference to lower-priority tasks.
2. Priority inversions, which may result when a higher-priority task needs a resource allocated to a lower-priority one, are bounded.
3. Non-schedulable activities, including both non-realtime activities and operating-system activities, don't exceed the remaining capacity in any particular division.

Because of condition 3, we must discount those activities outside of the control of the operating system, yielding the external factors provision above.

We can then derive the following operating system requirements (OSRs):

1. The OS must support fixed-priority preemptive scheduling for tasks.
2. The OS must provide priority inheritance or priority-ceiling emulation for synchronization primitives.
3. The OS kernel must be preemptible.
4. Interrupts must have a fixed upper bound on latency. By extension, nested interrupt support is required.
5. Operating-system services must execute at a priority determined by the client of the service:
    - All services on which it's dependent must inherit that priority.
    - Priority inversion avoidance must be applied to all shared resources used by the service.

OSR 3 and OSR 4 impose a fixed upper bound on the latency imposed on the onset of any particular realtime activity. OSR 5 ensures that operating system services themselves—which are internal factors—don't introduce non-schedulable activities into the system that could violate basic requirement 3.

How does an RTOS differ from a conventional OS?

The key characteristic that separates an RTOS from a conventional OS is the predictability that's inherent in all of the requirements above. A conventional OS, such as Linux, attempts to use a "fairness" policy in scheduling threads and processes to a CPU. This gives all applications in the system a chance to make progress, but doesn't establish the supremacy of realtime threads in the system or preserve their relative priorities, as is required to guarantee that they'll finish on time. Likewise, all priority information is usually lost when a system service, usually performed in a kernel call, is being performed on behalf of the client thread. This results in unpredictable delays preventing an activity from completing on time.

By contrast, the microkernel architecture used in the QNX OS is designed to deal directly with all of these requirements.

The microkernel itself simply manages threads within the system and allows them to communicate with each other. Scheduling is always performed at the thread level, and threads are always scheduled according to their fixed priority—or, in the case of priority inversion, by the priority as adjusted by the microkernel to compensate for

priority inversions. A high-priority thread that becomes ready to run can preempt a lower-priority thread. This action is known as *thread displacement* and it becomes complex in multicore systems. For more information, refer to the "Thread displacement on multicore systems" section in the *Programmer's Guide*.

Within this framework, all device drivers and operating system services apart from basic scheduling and interprocess communication (IPC) exist as separate processes within the system. All services are accessed through a synchronous message-passing IPC mechanism that allows the receiver to inherit the priority of the client. This priority-inheritance scheme allows OSR 5 to be met by carrying the priority of the original realtime activity into all service requests and subsequent device-driver requests.

There's an attendant flexibility available as well. Since OSR 1 and OSR 5 stress that device-driver requests need to operate in priority order, at the priority of the client, throughput for normal operations can be substantially reduced. Using this model, an operating service or device driver can be swapped out in favor of a realtime version that satisfies these requirements. Complex systems generally partition such resources into realtime and non-realtime with different service and device-driver implementations for each resource.

Because of the above, all activities in the system are performed at a priority determined by the thread on whose behalf they're operating.

What is a soft realtime OS?

A soft realtime OS must be capable of doing effectively everything that a hard realtime OS must do. In addition, a soft realtime OS must be capable of providing monitoring capabilities with accurate cost accounting on the tasks in the system. It must determine when activities have failed to complete on time or when they have exceeded their allocated CPU capacity, and trigger the appropriate response.

How does all of this affect my application?

If you're writing an application or system for deployment on a realtime OS, it's important to consider the effect that the RTOS characteristics have on the execution of the application, and to understand how it can be used to your benefit. For example, with an RTOS you can increase responsiveness of certain operations initiated by the user.

Most applications normally run at the default user priority within the system. This means that applications normally compete with each other for CPU capacity. Without the type of realtime schedule mentioned above, you can manipulate the priorities of the processes in the system to have certain activities run preferentially to others in the system. Manipulation of the priorities is a double-edged sword. Used judiciously, it can dramatically improve response in areas that are important to the user. At the same time, it's possible to starve other processes in the system in a way that typically doesn't happen on a conventional desktop system.

The key to ensuring that higher-priority processes and threads don't starve out other processes is to be certain of the limits imposed on their execution. By pacing the execution, or by throttling it in response to load, you can limit the CPU time consumed by these activities so user processes get their share of CPU time.

Media players, such as audio players (MP3, **.wav**, etc.) and video (MPEG-2), are a good example of applications that can benefit from priority manipulation. The operation of a media player can be tied to the media rate that's required for proper playback (i.e., 44 kHz audio, 30 fps video). So within this constraint, a reading thread that buffers data and a rendering or playback thread can both be designed to awaken on a programmable timer, buffer or render a single frame, and then go to sleep awaiting the next timer trigger. This provides the necessary pacing, so that the priority can be assigned above normal user activities, but below more critical system functions.

By choosing appropriate priorities, you can ensure that playback occurs consistently at the given media rate. A well-written media player also takes into account quality of service, so that if it doesn't receive adequate CPU time, it can reduce its requirements by selectively dropping samples or follow an appropriate fall-back strategy. This then prevents it from starving other processes as well.

You may also wish to treat certain user events preferentially within the system. This works well when you increase the concurrency within an application, and when the event can always be handled in a predictable, small amount of time. The key concern here is the frequency at which these events can be generated. If they can't occur too frequently, it's safe to raise the priority of the thread responding to them. If they can occur too frequently, other activities will be starved under overload conditions.

The simplest solution is to divide responsibility for events into different handling threads with different priorities and to queue requests or deliver them with messages. You can tie the handler's execution to a timer, so that the execution of the thread is throttled by the timer, handling a fixed number of requests within a given interval. This stresses the importance of factoring areas of application responsibility, giving a flexible design with opportunities for effective use of concurrency and preferential response, all of which lead to a greater feel of responsiveness.